Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, government and non–governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee ISO/IEC JTC1. Draft International Standards adopted by the joint technical committees are circulated to the national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

ISO/IEC 14478–3 was prepared by Joint Technical Committee ISO/IEC JTC1, *Information technology*, Subcommittee SC24, *Computer graphics and image processing.*

This International Standard currently consists of the following four parts under the general title *Information technology — Computer graphics and image processing — Presentation environments for multimedia objects (PREMO):*

— *Part 1: Fundamentals of PREMO*

— *Part 2: Foundation Component*

— *Part 3: Multimedia Systems Services*

— *Part 4: Modelling, Rendering, and Interaction Component*

Annex A forms an integral part of this part of ISO/IEC 14478. Annexes B to D are for information only.

Introduction

The Multimedia Systems Services (MSS) component of PREMO provides a standard set of services that can be used by multimedia application developers in a variety of computing environments. Enabling multimedia applications in a heterogeneous, distributed computing environment is the design motivation for the MSS. This is an increasingly prevalent computing model, and a solution that meets the needs of this environment can more easily be scaled to stand–alone systems than vice versa.

The principal reasons for defining the MSS are:

a) provide abstractions and mechanisms that make it possible for applications to deal with the problems of distributed multimedia computing successfully;

b) facilitate the implementation of complex applications, such as video conferencing;

c) provide abstractions that make it possible for applications to deal with media devices without regard to specific characteristics of the platform, attached devices, or the network(s) connecting the platforms and devices;

d) to provide a standard methodology, especially for handling “live” data;

e) insure scalability to large organizations;

f) insure adequate performance in adverse conditions;

g) facilitate Quality of Service commitments; and

h) consider the time critical nature of the data.

The primary goal of the MSS is to provide an infrastructure for building multimedia computing platforms that support interactive multimedia applications dealing with synchronized, time–based media in a heterogeneous distributed environment. Operation in a distributed environment is important because of significant trends in the computer industry towards client/server and collaborative computing. Another significant trend is towards multimedia enabled computing. The inevitable result will be an intersection of these trends to produce a distributed multimedia environment with a topology similar to Figure 1.

The MSS is intended to address a broad range of application needs. It extends the multimedia capabilities of stand-alone computers to capabilities that are usable both locally and remotely. The Multimedia Systems Services gives applications the ability to handle:

i) live data remotely;

j) stored data remotely;

k) both live and stored data simultaneously;

l) multiple kinds of data simultaneously; and

m) new kinds of devices and media types.

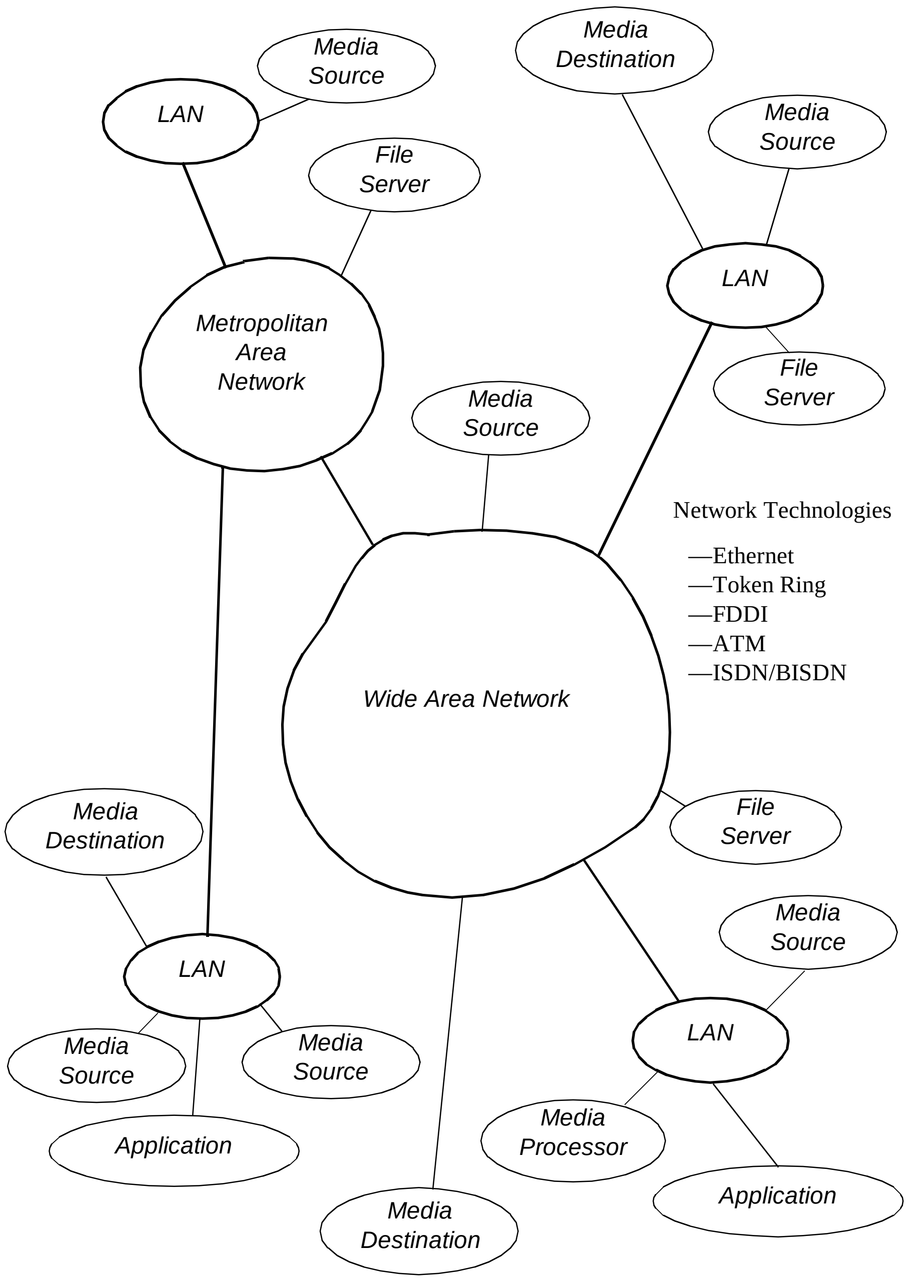


Figure 1 — Distributed multimedia environment

To provide support for remote media device control and remote media access that derive from the above application scenarios, the Multimedia System Services uses two distinct mechanisms. To support interaction with remote objects, the Multimedia Systems Services depends upon an underlying object model and infrastructure, as described in ISO/IEC 14478–1 (PREMO). To support the media independent streaming of time critical data, the Multimedia Systems Services defines a Media Stream Control.

The MSS does not address:

n) encryption and security;

o) intellectual property rights and accounting;

p) scripting;

q) user interfaces; or

r) sharing of data between applications.

Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 3: Multimedia Systems Services

1 Scope

This part of ISO/IEC 14478 defines a standard set of multimedia system services that can be used by multimedia application developers in a variety of computing environments. The focus is on enabling multimedia applications in a heterogeneous, distributed computing environment. Throughout this part of ISO/IEC 14478, this component will also be referred to as "Multimedia Systems Services", and abbreviated as MSS.

The Multimedia Systems Services constitutes a framework of "middleware" — system software components lying in the region between the generic operating system and specific applications. As middleware, the Multimedia Systems Services marshals lower–level system resources to the task of supporting multimedia processing, providing a set of common services which can be used by multimedia application developers.

The Multimedia Systems Services encompasses the following characteristics:

a) provision of an abstract type for a media processing node, extensible through subtyping to support abstractions of real media processing hardware or software;

b) provision of an abstract type for the data flow path or the connection between media processing nodes, encapsulating low–level connection and transport semantics;

c) grouping of multiple processing nodes and connections into a single unit for purposes of resource reservation and stream control;

d) provision of a media dataflow abstraction, with support for a variety of position, time and/or synchronization capabilities;

e) separation of the media format abstractions from the dataflow abstraction;

f) synchronous exceptions and asynchronous events;

g) application visible characterization of object capabilities;

h) registration of objects in a distributed environment by location and capabilities;

i) retrieval of objects in a distributed environment by location and constraints;

j) definition of a Media Stream Protocol to support media independent transport and synchronization.

The Multimedia Systems Services rely on the object model of ISO/IEC 14478-1 (Fundamentals of PREMO) and the object types and non–object data types defined in ISO/IEC 14478-2 (PREMO Foundation Component).

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 14478. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of this international standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 14478-1:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 1: Fundamentals of PREMO.*

ISO/IEC 14478-2:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 2: Foundation Component.*

ISO/IEC 14478-4:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 4: Modelling, Rendering, and Interaction Component.*

ISO/IEC 10918-1:1994, *Information technology — Digital Compression and Coding of Continuous–Tone Still Images (JPEG).*

ISO/IEC 11172:1992, *Information technology — Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5Mbit/s (MPEG).*

3 Definitions

3.1 PREMO Part 1 definitions

This part of ISO/IEC 14478 makes use of all terms defined in ISO/IEC 14478-1 (Fundamentals of PREMO).

3.2 PREMO Part 2 definitions

This part of ISO/IEC 14478 makes use of all terms defined in ISO/IEC 14478-2 (PREMO Foundation Component).

3.3 Additional definitions

For the purposes of this part of ISO/IEC 14478, the following definitions apply.

3.2.1 configuration objects: Collective name for format, quality of service descriptor, and media stream protocol objects.

3.2.2 jitter: Delay variance.

3.2.3 processing element (for virtual devices): Conceptual entity describing the internal behaviour of virtual device objects.

3.2.4 virtual connection adapter: Conceptual entity describing the configuration process performed by a virtual connection object.

3.2.5 unicast connection: One–to–one connection; an output port may be connected to one input port only, and an input port may be connected to one output port only.

3.2.6 multicast connection: One–to–many connection; an output port may be connected to several input ports, and an input port may be connected to several output ports.

The following alphabetical list gives the subclause of each definition.

configuration objects	3.2.1
jitter	3.2.2

multicast connection	3.2.6
processing element (for virtual devices)	3.2.3
unicast connection	3.2.5
virtual connection adapter	3.2.4

4 Symbols and abbreviations

ATM:	Asynchronous Transfer Mode.
CATV:	Cable TV
DMA:	Direct Memory Access.
FSM:	Finite State Machine.
IEC:	International Electrotechnical Commission.
IS:	International Standard.
ISO:	International Organization for Standardization.
JPEG:	Joint Picture Experts Group.
LAN:	Local Area Network
MIDI:	Musical Instrument Digital Interface
MPEG:	Moving Picture Experts Group.
MSS:	Multimedia Systems Services
PREMO:	Presentation Environment for Multimedia Objects.
QoS:	Quality of Service.
TCP:	Transmission Control Protocol.
UDP:	User Data Protocol.
RTP:	Real–Time Protocol

5 Conformance

A conforming implementation of the PREMO Multimedia Systems Services shall comply with the general conformance rules defined in clause 5 of ISO/IEC 14478-1 and the component specification in clause 11 of This part of ISO/IEC 14478.

6 Overview of the Multimedia Systems Services

6.1 Introduction

This clause presents several comprehensive views of the Multimedia Systems Services, which, taken together, represent a broad, architectural summary. These views include:

a) an object interaction diagram, which characterizes the dynamic relationships among instantiated objects and to illustrate client visible interfaces;

b) a subtyping diagram, which describes the subtyping hierarchy among MSS objects;

c) a short description of the life cycle of Multimedia Systems Services objects.

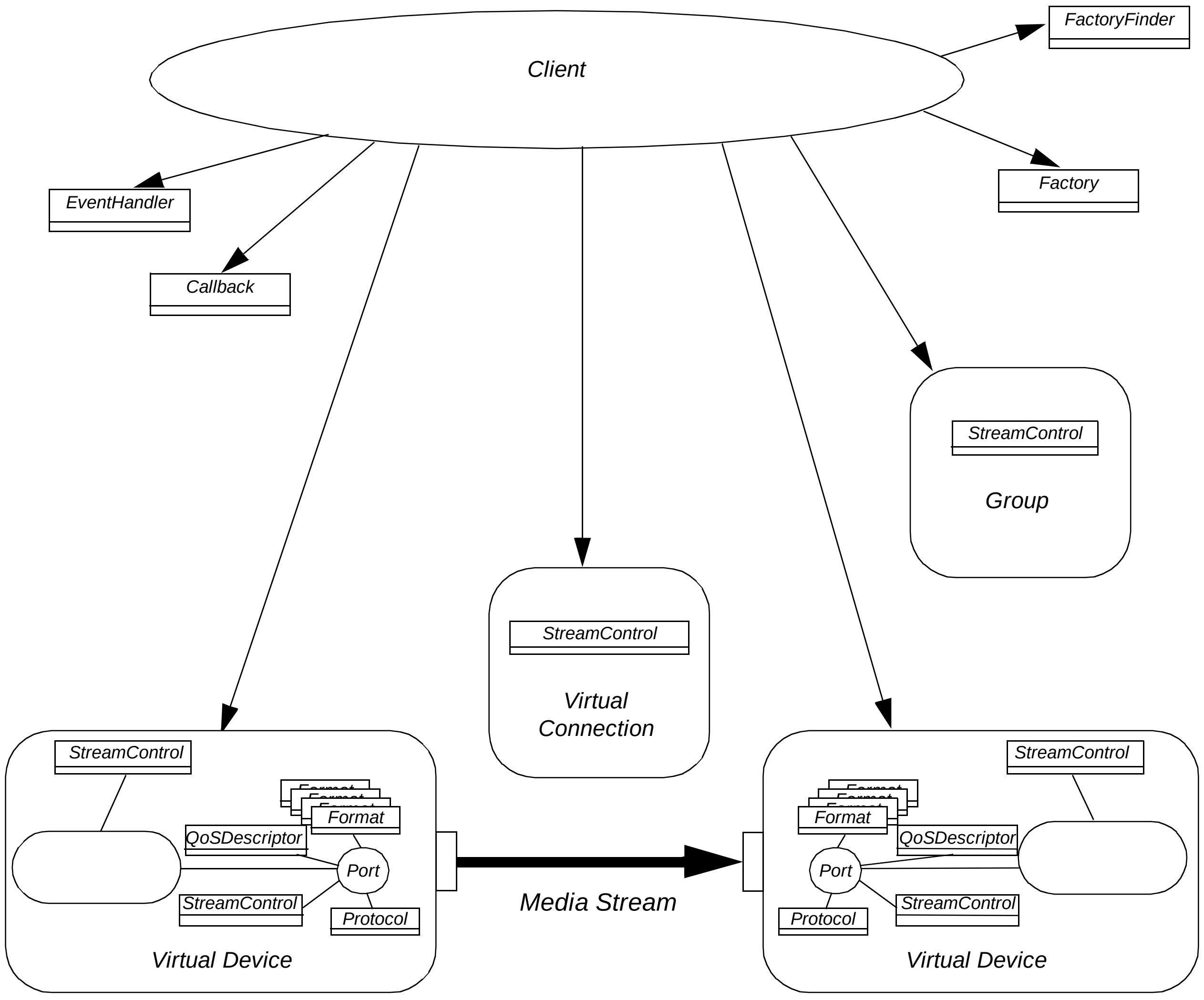


Figure 2 — Multimedia system services client interaction

A somewhat larger example of how the various MSS objects may be used is also given in Annex B.

6.2 Object framework

Figure 2 on page 4 summarizes the interactions between Multimedia Systems Services framework objects and a client; Figure 3 on page 6 summarizes the interaction among framework objects. As seen in Figure 2, only a subset of the objects and interfaces are actually visible to a client of MSS. In particular, much of the interaction between the virtual connection and other objects in the framework is not client visible. This part of ISO/IEC 14478 is concerned primarily with client visible interfaces; implementations may extend these interfaces for implementation–specific reasons, to comply with the behaviour of the objects, as defined in this standard.

Figure 2 is suggestive, rather than realistic, since the objects shown are abstract types, rather than concrete types which would normally be instantiated. Also, object creation and destruction are not shown in this diagram.

In Figure 2, the client is communicating with a small dataflow graph, comprised of two virtual devices and a virtual connection. A group object, which assists the client, is also shown. The client interacts with the objects indicated by the arrows. Each of these objects may be local or remote.

Each virtual device is a processing node in the dataflow graph. The nature of the processing (capture, encoding, filtering, etc.) varies according to the specific object (and is implemented by subtyping). Each virtual device contains a stream control object and one or more format objects shown by the boxes in the shaded areas. Virtual connections and groups also have an associated stream object and this association is represented similarly. These associations are referred to as inclusion. Although not explicitly shown in the diagram, the client interacts directly with the included stream and format interfaces.

A stream control object provides the client with an interface to observe media stream position in various terms (as a function of media transport, media samples, or logical time; see clause 8). Stream control objects also provide synchronization operations. Stream control objects do *not* perform the effective transfer of media data; they merely act as an entry for the clients to control media flow.

In addition to a stream control, a virtual device also contains one or more ports, describing an input or output mechanism for the virtual device. Ports are framework abstractions that do not have a client–visible interface. Virtual devices provide operations to select a specific port, using an abstract non–object data type as an opaque handle.

Just as the stream control object allows a media stream control abstraction which is separable from media processing, the format object provides an abstraction of the details of media formatting which is separate from both processing and flow–control. For example, the details of a frame–dependent video encoding, like MPEG, would be represented by a subtype of format.

The virtual connection provides operations to create a connection between an output port of one virtual device and an input port of another, fully encapsulating low–level transport semantics. Virtual connections also provide support for multicast connections. An included stream object provides operations for controlling the dataflow on the virtual connection.

The group object, shown in Figure 2, provides assistance to the client to manage the dataflow graph of the two virtual devices and the virtual connection. A group object provides a convenient mechanism for atomic resource allocation and specification of end–to–end Quality of Service (QoS) values for the whole graph. The group object gives access to a stream object, through which the client can control dataflow for the encapsulated graph.

Multimedia Systems Services objects are instantiated by factories. A factory provides the client facilities to select among the various objects that the factory is capable of creating. A client can also use the factory finder service to find a reference to a factory capable of instantiating an object whose properties satisfy a list of constraints.

A client can register interest in receiving specific events produced by the various objects. This is done using the callback mechanism and the event model as described in ISO/IEC 14478-2 (part 2 of PREMO).

Figure 3 shows the internal connections between MSS objects. For the most part, the client is unaware that these connections exist, and this part of ISO/IEC 14478 will not focus on the low–level details of these connections and their relevant interfaces. They are shown here to help explain the Multimedia Systems Services architecture. The primary purpose of most of the internal connections is to off–load work from the client. Note, for example, that the virtual connection interacts with the formats of both the source and target virtual devices.This allows the virtual connection to match those formats without client intervention. The group and the stream associated with the group provide similar assistance to the client; the group can assist in resource allocation, while the associated stream can assists in stream control. The dashed arrows show that the objects send events to the client via the event handler.

6.3 Subtyping diagram

Another view of the MSS Architecture is given in Figure 4 on page 7. This is a simplified subtyping diagram for the MSS objects which does not contain all object types defined in this part of ISO/IEC 14478 neither their full type graph; the reader should refer to Annex A for a more complete diagram.

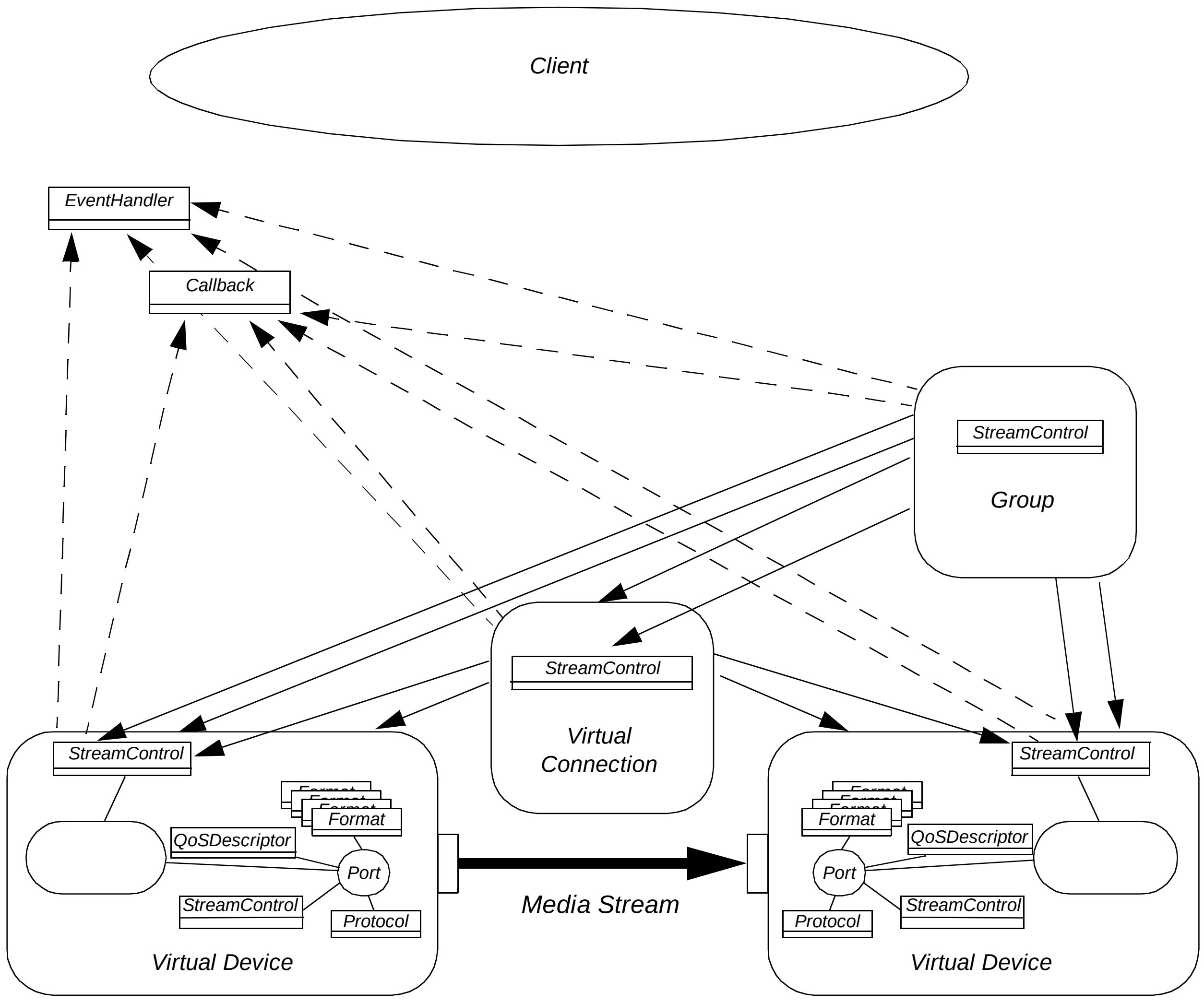


Figure 3 — Multimedia system services internal interfaces

The types in the diagram will be discussed in detail elsewhere in this part of ISO/IEC 14478. For the time being the only important characteristic to note is that all types defined in MSS are subtypes of *EnhancedPREMOObject* (see ISO/IEC 14478-2), thereby inheriting all features described in the Foundation Component of PREMO. Also, all objects are subtypes of *PropertyInquiry*, which is further described in 8.1.2 of ISO/IEC 14478-2 (part 2 of PREMO).

6.4 MSS object life cycle

Figure 5 on page 7 gives a schematic view of the life cycle of an MSS object and its reference. Beyond the basic life cycle of PREMO objects, as described in ISO/IEC 14478-1, all MSS objects rely on the concepts of factories and factory finders defined in the extended profile of ISO/IEC 14478-2 (see 8.2.1 of ISO/IEC 14478-2 for a detailed description of these facilities), which encapsulate normal object and object reference creation. The client has the ability to create objects through these factories by specifying constraints on the capabilities of the object to be created (e.g., media formats it can process, quality of service it can provide, etc.). The client receives from the factory an object reference to an object obeying these requirements. This object reference goes then through the life cycle as described in 8.11 of ISO/IEC 14478-1.

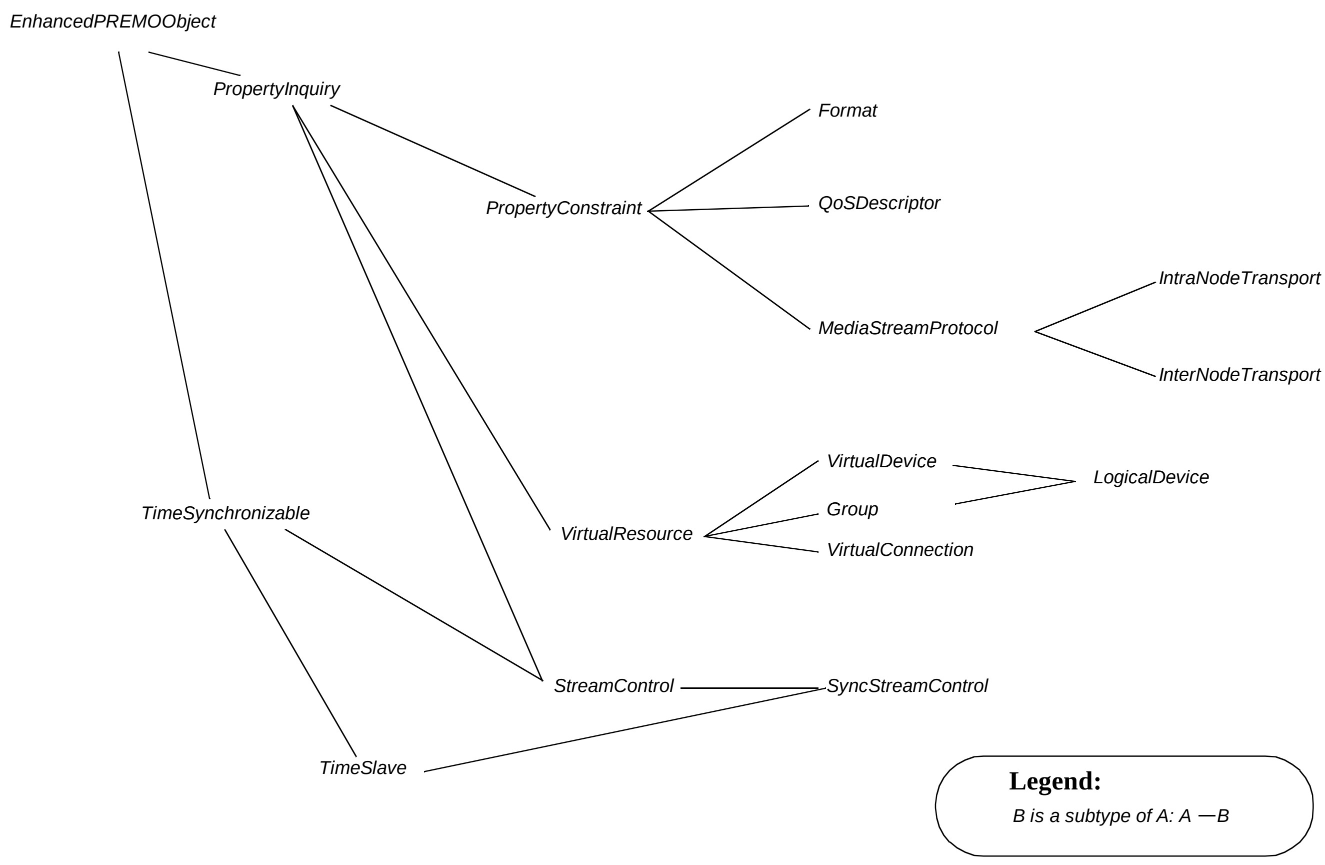
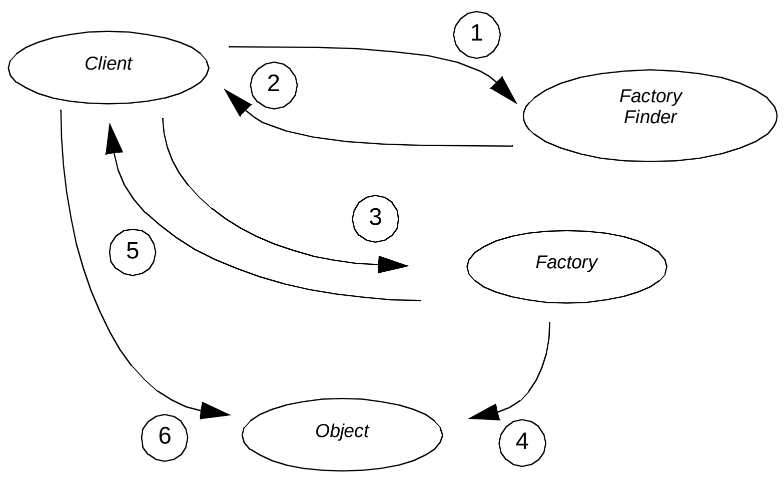


Figure 4 — Subtyping diagram



1. *Client requests a reference to a factory capable of satisfying a capability list passed as parameter.*
2. *Factory Finder returns a Factory reference*
3. *Client requests the creation of an object from the Factory, with a constraint list on the object to be created.*
4. *Factory possibly creates the object…*
5. *…and returns the reference to its client.*
6. *Later, the client destroys the reference, which eventually leads to the destruction of the object.*

Figure 5 — MSS Object life cycle

7 Configuration objects

7.1 Introduction

The various MSS object types, categorized as *configuration* objects, are used as information depositories for other objects. Configuration objects do not provide complicated behaviour to their clients; instead, their role is to act as placeholders for the necessary parameters which allow other objects to function properly. Configuration objects are not used in isolation; they are always associated with an object of type *VirtualResource* object (see 9.1) whose behaviour they affect.

All configuration objects are subtypes of *PropertyConstraint* objects (see 8.1.3 of ISO/IEC 14478-2), and the various parameters they provide to other objects are stored and manipulated through the property constraining mechanism which characterize the *PropertyConstraint* objects. Typically, configurable MSS objects (e.g., resource objects, see clause 9) contain several instances of configuration objects, whose references can be accessed by the external clients; through the *constrain*, *select*, etc., operations these clients can then set the required values for the configuration objects.

Usage of configuration objects is the basic mechanism for configurability in PREMO. Clients may inquire the key–value pairs associated with these objects and, through the property constrain and selection mechanism (see 8.1.3 of ISO/IEC 14478-2) may restrict these values. The algorithms for constraining the possible values (within the limits defined by the capabilities and the native property values of the configuration objects) depend on the client and is not standardized by PREMO. These algorithms may take into consideration the full set of configuration objects associated to the same *VirtualResource* instance. Note that further subtypes of configuration objects may be defined with an inherent constraint algorithm, which is made available through the *select* operation of the configuration object (see again 8.1.3 of ISO/IEC 14478-2).

The MSS has three categories of configuration objects, described below:

a) Format objects;

b) Transport and Media Stream Protocol objects;

c) Quality of Service objects.

7.2 Format objects

The role of the *Format* object type is to represent the details of the media format (that is, the organization of the bitstream) at a particular port of a device (see clause 9). The characteristics of a specific media format are described in form of object properties defined on the corresponding *Format* object.

NOTE — For example, the organization of data in an MPEG data stream is different from the organization of data in an a-law audio stream. The *Format* object allows the client to specify as little or as much about a particular encoding as it wants. When a connection is made to a port, the virtual connection interacts with the *Format* object to negotiate the details of the bitstream to be passed between virtual device ports.

The Multimedia Systems Services do not define specific format objects; instead, it defines a general architecture which makes such a specification possible. However, the (informative) Annex C of this part does describe a hierarchy of format and device objects which provide a good example of how this general architecture should be used.

7.3 Transport and Media Stream Protocol objects

When a virtual connection determines that two virtual devices cannot be directly connected (often because they are on different machines), it creates a virtual connection adapter to transport media data between them (see 9.3). The virtual devices may reside in different implementations of the Multimedia Systems Services, so the virtual connection adapter must share a common protocol in order to interoperate.

The purpose of the Transport and Media Stream Protocol (MSP) objects is to provide information on how media data is conveyed among processing nodes. It is not the role of MSS to give a detailed specification of the various possible communication protocols, only references to existing protocol specifications are made here. In MSS, MSP objects have two subtypes:

a) *IntraNodeTransport*, which refers to communication among nodes taking place through shared memory (e.g., two nodes processing in the same address space of a workstation using, e.g., DMA)

b) *InterNodeTransport*, which refers to communication among nodes taking place over a network, or through some inter–process communication means. The various protocol names which can characterize these protocols include IPC, UDP, RTP, ATM, or NETBIOS.

The virtual connection determines the format of the media data, including its bit–level representation, by negotiation with the relevant *Format* objects (see 7.2) and perhaps with client intervention. Similarly, the client will conduct negotiations on the detail of the transfer protocols, using the MSP objects. From the point of view of MSP objects the media data is an opaque entity. Neither the virtual connection adapter nor the underlying network transports know how to extract information from the media data.

To allow for future revisions, the MSP properties include a protocol version number.

7.4 Quality of Service Descriptor objects

In order for a virtual resource to be useful, it must obtain the physical resources required for it to do its job. Resources include both system resources that are typically not multimedia specific, such as those provided by the CPU, memory, and network subsystems, as well as specialized multimedia resources such as audio and video devices.

Because the quality of service that can be provided by many resources varies considerably, the client must also specify the desired quality of service when requesting a resource; this is done by setting the properties of a special information depository object in MSS, called the *QoSDescriptor* object.

Though quality of service can take on many meanings, many of them media and device specific, the Multimedia Systems Services defines a core set of *QoSDescriptor* properties that can be used by a client to specify the quality of service of interest. The core *QoSDescriptor* characteristics defined by the Multimedia Systems Services are:

a) guaranteed level (key “*GuaranteedLevelK*”): provides options for “*Guaranteed*” service, “*BestEffort*” service or “*NoGuarantee*” service;

b) reliable (key “*ReliableK*”): the delivery of data is reliable or not;

c) delay bounds (key “*DelayBoundsK*”): the minimum and maximum delay;

d) jitter bounds (key “*JitterBoundsK*”): the minimum and maximum jitter (delay variance);

e) bandwidth bounds (key “*BandwidthBoundsK*”): the minimum and maximum bandwidth.

The client specifies the desired *QoSDescriptor* properties by accessing the quality of service information from a *VirtualResource* object, and manipulating the property values with the property management operations.

The guaranteed level property is an indication of the performance required of the virtual connection (see 9.3 for a more detailed presentation of virtual connections). A “*Guaranteed*” connection will reserve resources appropriate to handle worst–case needs for the media transfer in order to make sure that the data always arrives and is on time. A “*BestEffort*” connection will provide the best possible performance while using optimistic amounts of resources. This may produce situations where the data occasionally arrives late. “*NoGuarantee*” uses the minimum amount of resources for the connection and do as well as possible.

The reliability property indicates whether the application is willing to suffer data loss.

Delay is the amount of time between the transmission of the data and the receipt of the data. Different applications will have different requirements. For instance, an audio conference would be unwilling to live with a 2 second delay, whereas a non-interactive video playback application might find it acceptable.

Bandwidth is the amount of data per unit time that the connection will be required to support or, in the case of an input port, to expect. For example, a video conference might require 384 Kbits/second while an MPEG stream might require 1.5 Mbits/second. By defining the range of the bandwidth required, the connection will understand the maximum burst that it must handle.

Jitter is the amount of delay variance. For example, an ISDN channel that presents a "slot" of data every 125 microseconds has a jitter of 0, since there is no variance in the arrival time of the data. If an application requests a jitter close to 0, then the connection will try to find an isochronous network connection between the two virtual devices.

Subtypes of *QoSDescriptor* objects may extend the *QoSDescriptor* management with additional characteristics.

8 Stream Controls

The *StreamControl* type, and its subtypes, provide a single point of focus for all inquiry and control of media stream progress in a media type independent way. *StreamControl* objects are never created in isolation; they are included objects of *VirtualResource* objects whose role is to monitor and control stream progress for the overall resource. Various virtual resource objects need subtypes of the stream objects defined in MSS to cope with the various media–dependent characteristics, e.g., details of media presentation. The MSS gives a formal specification of two objects only, namely the *StreamControl* and the *SyncStreamControl* object types; semantic details of the various subtypes are beyond the scope of MSS and are not considered in This part of ISO/IEC 14478.

NOTE — Further PREMO components may define subtypes of the *StreamControl* and *SyncStreamControl* objects.

The *StreamControl* type provides operations to observe and control media position. A complication is that different objects require different concepts of position:

a) *Transport aware object*: the object might understand transport packets, but not understand the structure of the media stream. The object can only report the stream address, that is the byte count since the stream began to flow.

b) *Media Stream aware object:* the object might understand media samples, but not how media samples translate into stream time. The object can report the stream sample count.

c) *Time aware object:* the object understands how to extract stream time from the media stream. The object can report the stream time.

To answer to these concerns, the *StreamControl* object is defined in MSS to be a subtype *TimeSynchronizable*, described in detail in 7.9 of ISO/IEC 14478-2. Inheriting the behaviour of these objects, clients of *StreamControl* objects have the ability to access and control the stream position in terms of:

d) *Media stream data*: The internal progression space of the object (i.e., index of a sample).

e) *Relative time*: The time elapsed from an origin, settable by the client. One can achieve effects like "pause at 100ms from origin".

Synchronization elements can be put at various points of the stream, using either media stream data or relative time. This means that a *StreamControl* object will send events when these points are crossed. Such synchronization elements can be set periodically, i.e., one can achieve effects like "send an event at each 100ms".

Stream control objects do *not* perform the transfer of media data; they merely act as an entry for the clients to control media flow. Conceptually, media data transfer is done by the processing element of the *VirtualDevice* object and, hence, is opaque to the client.

The *StreamControl* object does not specify further the internal progression space of *TimeSynchronizable* (which is a generic type), i.e., whether the internal progression is done in the integer, real, or time domain. Specific *VirtualDevice* objects have to actualize the *StreamControl* objects they use by making this decision.

8.1 *StreamControl* objects

The *StreamControl* object adds a finer media control to its supertype *TimeSynchronizable*. This finer control is achieved through the refinement of the finite state machine governing the behaviour of *TimeSynchronizable*.

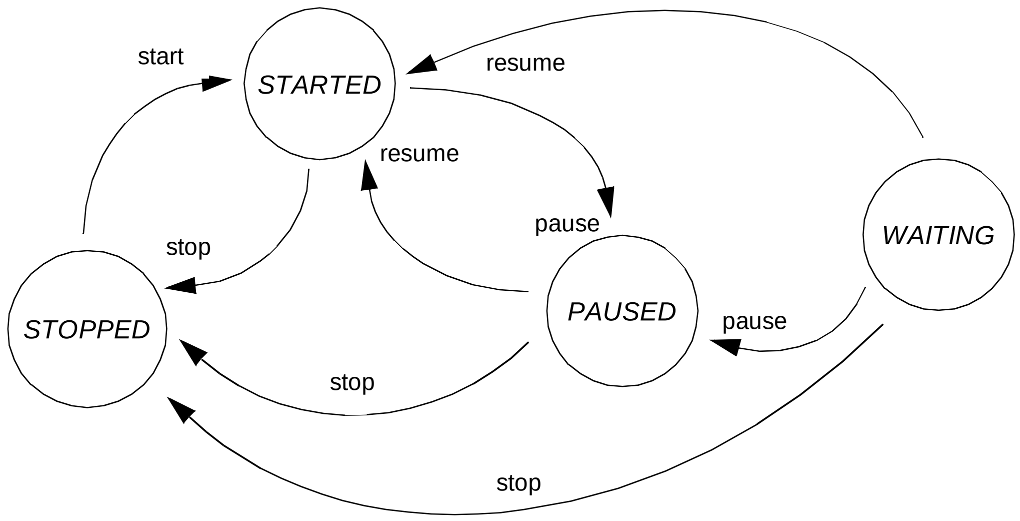


Figure 6 — State transition diagram for a *TimeSynchronizable* object

Figure 6 depicts the finite state machine of *TimeSynchronizable*, as defined in ISO/IEC 14478-2 (not all state transition operations are present on the figure, only the most important ones); for a detailed discussion on the behaviour of *TimeSynchronizable*, see 7.9 (page 12) of ISO/IEC 14478-2. Figure 6 gives a rough overview of the finite state machine of a *StreamControl* object, again omitting some details (the detailed state transition table is in 10.6.1).

The *StreamControl* object adds three new states to the state machine of *TimeSynchronizable*, namely *MUTED*, *PRIMING*, and *DRAINING*. Three new operations are also defined, which control the state transitions to and from these new states: *mute*, *prime*, and *drain*.

MUTED and *PRIMING* are refinements of the *STARTED* state of *TimeSynchronizable*. The additional semantics in these states is related to the notion of data presentation. In describing the *STARTED* state, ISO/IEC 14478-2 refers to data presentation in very general terms only, as one abstract processing step of the object at that level of abstraction. This abstract step is performed through the invocation of a protected operation, called *processData*. The specification leaves the semantic details of what presentation means to the various subtypes of *Synchronizable*. Although the *StreamControl* object does not specify what presentation means either (and leaves the details to the subtypes of *StreamControl*), the specification of *MUTED* and *PRIMING* gives a somewhat finer control on the behaviour of the *StreamControl* object with respect to presentation. This refinement is as follows:

— *MUTED*: no presentation occurs while the object is in this state, and the media data are disregarded. In other words, progression on the stream occurs (i.e., all synchronization actions are performed) but the *processData* operation, which represents the abstract notion of processing media data, is not invoked. The operation *mute* is analogous to the operation *start*, inherited from *TimeSynchronizable*, the only difference being the target state of the transition (*MUTED* instead of *STARTED*).

— *PRIMING*: no presentation occurs while the object is in this state, and the media data are buffered in an internal buffer. In other words, progression on the stream occurs (i.e., all synchronization actions are performed) and the media data are stored internally instead of being presented, i.e., instead of calling the *processData* operation. The operation *prime* is analogous to the operation *start*, inherited from *TimeSynchronizable*, the only difference being the target state of the transition (*PRIMING* instead of *STARTED*). If the internal buffer of the object is full, i.e., no stream data can be stored any more, the object makes an internal state transition to *PAUSED*.

The third additional state, *DRAINING*, is the counterpart of *PRIMING* in buffer control. When set to this state, the object empties the buffer filled up by a previous *PRIMING* state; when the buffer is empty, the object makes an internal state transition to *PAUSED*. While emptying the buffer, presentation of data also occurs. The operation *drain* is defined to set the *StreamControl* object into *DRAINING* state.

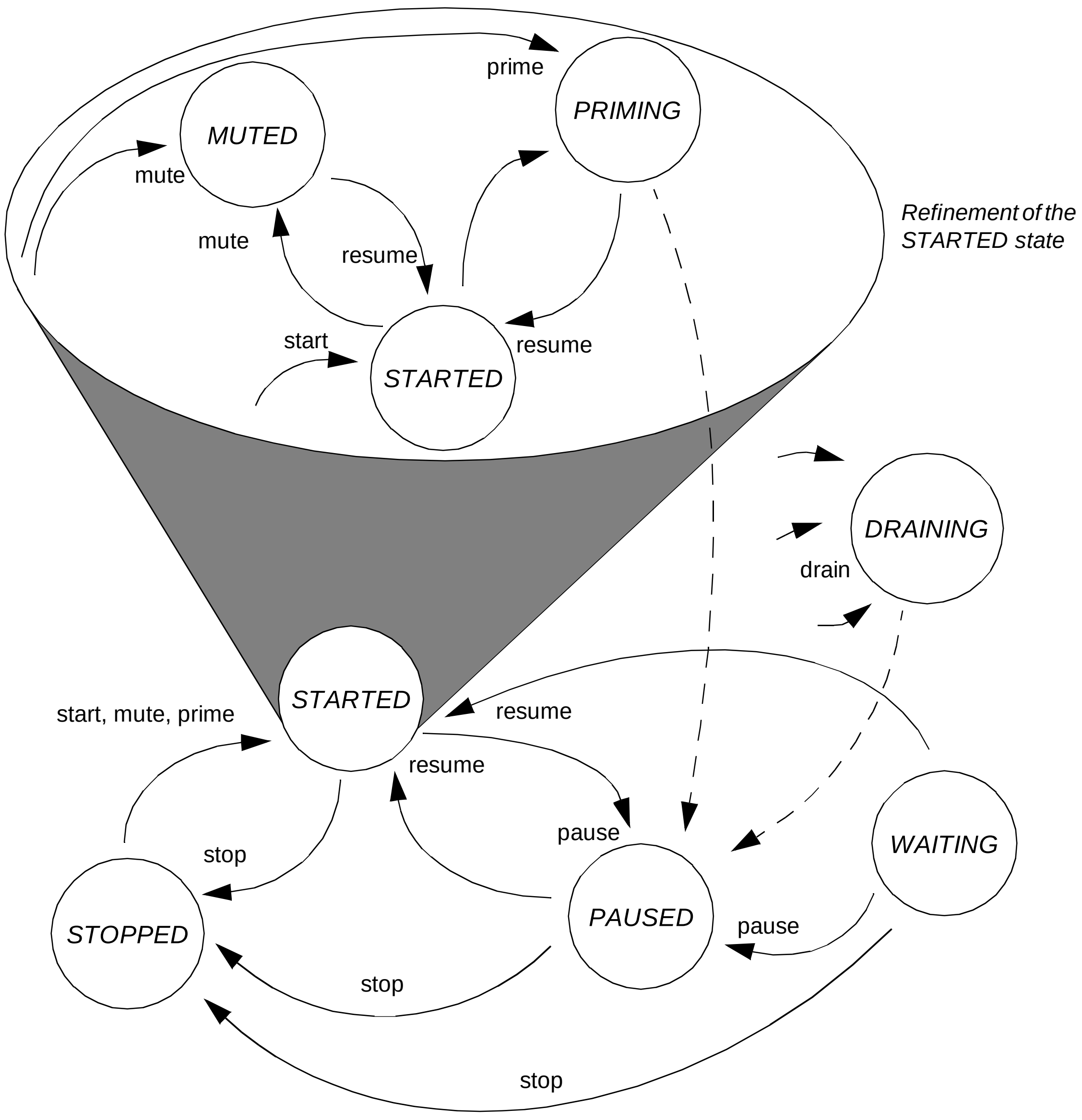


Figure 7 — State transition diagram for a *StreamControl* object

A transition to *STARTED* state from both *PRIMING* and *DRAINING* states is possible; conceptually, the internal buffers are to be instantaneously emptied before the normal media flow is resumed.

Note that, as a subtype of *Synchronizable*, the *StreamControl* object also inherits the possibility to have state transitions monitored by other objects (using the callback mechanism, see also 7.9.1.2 of ISO/IEC 14478-2). In particular, clients can be notified if internal buffers become full or empty while priming, respectively draining (in both cases an internal state transition to the state *PAUSED* takes place, which may be monitored).

Subtypes of *StreamControl* may add additional semantics to buffer control. As a typical case, if the streams are aware of their position within a dataflow network, some of the operations, like *prime* or *drain*, may also generate private control flow among the streams in this network. For example, *prime* on a *StreamControl* may also generate a control information to the *StreamControl* object "up–stream", i.e., the stream providing the data. Whether such additional protocol is defined or not depends on the subtypes of the *StreamControl* object.

StreamControl objects are also subtype of *PropertyInquiry*. This means that subtypes of *StreamControl* objects may also take part in complex negotiations based on properties, as described in 8.1 of ISO/IEC 14478-2 (part 2 of PREMO).

8.2 *SyncStreamControl* objects

The *SyncStreamControl* type is designed to permit the synchronization of multiple media streams. The client specifies a second *StreamControl* object to provide a master position reference to the *SyncStreamControl*. This functionality is achieved by inheriting the behaviour of the *TimeSlave* objects, defined in 7.9.3 of ISO/IEC 14478-2. *SyncStreamControl* is defined as a (multiple) subtype of both the *StreamControl* and the *TimeSlave* object types, thereby refining the finite state machine of a *TimeSlave* object the same way as *StreamControl* objects refine the behaviour of *TimeSynchronizable* objects.

NOTE — The *SyncStreamControl* type provides the client with a variety of options with respect to setting up how synchronization is to be achieved. For example, a video display device can be synchronized with an audio display device by setting up the appropriate relationship between their associated *StreamControl* objects. Assuming the video device supports a *SyncStreamControl* type, the audio device's *StreamControl* object can be made the master of the video device's *SyncStreamControl* object.

In another situation, the client may want to synchronize two displays to a common time reference. In this case, both displays would have to support the *SyncStreamControl* type. The client would make the *StreamControl* object associated with the time reference the master of the *StreamControl* objects associated with both the display objects.

9 Devices, Resources

9.1 Virtual Resources

A *virtual resource* is an abstraction of a physical resource that provides the client developer a consistent programming model, independent of the details of specific implementations. The notion of a virtual resource makes applications more portable across a variety of systems, while at the same time making the transparent sharing of physical resources possible.

The Multimedia Systems Services defines four basic subtypes of virtual resources:

a) *virtual devices*, which abstract media processors;

b) *virtual connections*, which abstract connections between virtual devices;

c) *groups*, which provide a convenient way to interact with a collection of virtual devices and connections;

d) *logical devices*, which provide a possibility to build a hierarchy of virtual devices.

The MSS defines a *VirtualResource* object type, which serves as a common, abstract supertype for these three categories. This clause describes this object type in more details.

9.1.1 Configuration objects on virtual resources

The externally visible aspects of virtual resources are described primarily in terms of a set of associated configuration objects (although subtypes of *VirtualResource* add additional operations which modify the behaviour of virtual resources, too). These configuration objects are created and contained by the *VirtualResource* object instance, i.e., these objects are *not* created by an external client. Also, the object references of these configuration objects are not directly visible to the client; instead, the virtual device object publishes (through a property with key "*ConfigurationNamesK*") information on the *semantic name* (which is a string) and the type of the various configuration objects it includes. Although it is possible, through a special *resolve* operation, to get access to the real object reference of these configuration objects, most of the operations defined on the *VirtualResource* type and its subtypes are defined in terms of the semantic names of the configuration objects, rather than object references. This mechanism provides an extra protection against configuration errors and makes implementation of virtual resource objects more effective. Also, most of the application should be able to be properly configured with the configuration objects (and the setting of their properties) as provided by default by the virtual resource object. On the other hand, through the object reference returned by the *resolve* operation, clients can also implement more complicated configuration procedures, using all the facilities of property control described in 8.1 of ISO/IEC 14478-2.

The static properties of configuration objects (see clause 7) are bound by the *VirtualResource* object when acquiring resources (through the *VirtualResource.acquireResource* operation) by internally invoking the *bind* operation on all these configuration objects. Static properties are unbound when the resources of the *VirtualResource* object are released.

Virtual resources can also act as event sources; an attribute of type *RefEventHandler* is associated to the object, hence the client can set the event handler for the events raised by a specific virtual resource.

9.1.2 Stream control

Typically, virtual resources are involved in the generation, consumption, or transport of media data. The flow of media data through a device or across a connection can be thought of as a stream. In order to monitor or control the progress of the (possibly abstract) stream, an overall *StreamControl* object (see clause 8) is associated to a *VirtualResource* object. This stream control object is created and contained by the *VirtualResource* object instance, i.e., this object is *not* created by an external client.

A client can get a reference to the *StreamControl* object associated with a *VirtualResource* by accessing the *stream* attribute of the object. Once the client has obtained a reference to a *StreamControl* object, it may be able to `cast` its reference to one of the more capable *StreamControl* subtypes.

It should be noted that not all virtual resources have a notion of stream control. In these cases, the resource's *stream* attribute may have a value of *NULLObject* (this may occur, for example, when the *VirtualResource* in question is abstracting an external device such as a CATV tuner, which would have no notion of stream control).

9.1.3 Resource management

When a virtual resource is requested to acquire resources, one or more *resource managers*, which are responsible for managing the access to physical resources necessary to realize the virtual resource, get involved in the resource allocation process. Depending on the Multimedia Systems Services implementation, there may be one resource manager per managed resource, or there may be one resource manager per group of resources or per system. From the point of view of PREMO, these resource managers are purely conceptual entities, and this part of ISO/IEC 14478 does not contain a detailed specification of these. However, this abstraction is useful in describing the externally visible behaviour of virtual resource objects.

When the *VirtualResource.acquireResource* operation is executed, the *VirtualResource* object communicates with the appropriate resource manager(s) to request allocation to resources. The resource manager(s) which are contacted by the virtual resource are dependent on the type of virtual resource and the Multimedia Systems Services implementation; typically, virtual resources are created with the information necessary to contact the appropriate resource manager(s) at resource allocation time. Some resource managers may have a generic interface for specifying resources, essentially providing a direct reflection of the *acquireResource* operation. Other resource managers may provide a more specific interface appropriate to the resource being managed; virtual resources requesting the use of such resources would by their nature understand the necessary vocabulary for making resource requests.

The resource manager(s) may allow multiple virtual resources to share a given physical resource, so long as the desired configuration of the virtual resource, e.g., quality of service, can be met. See 9.1.4 below for details on quality of service management.

NOTE — Figure 8 on page 15 depicts a possible resource management configuration. As presented on the figure, resource managers may work in conjunction with a *resource policy agent,* whose task is to permit the user to get involved with the resource allocation decisions, much in the same way as a window manager allows the user to make window size and placement decisions in a window system. When a resource manager has had a resource policy agent registered with it, resource management requests are redirected to the resource policy agent. The resource policy agent may get the user involved in the process, and it then resubmits a potentially modified request on the client's behalf back to the resource manager. This mechanism is similar in style to that used in the X Window System.

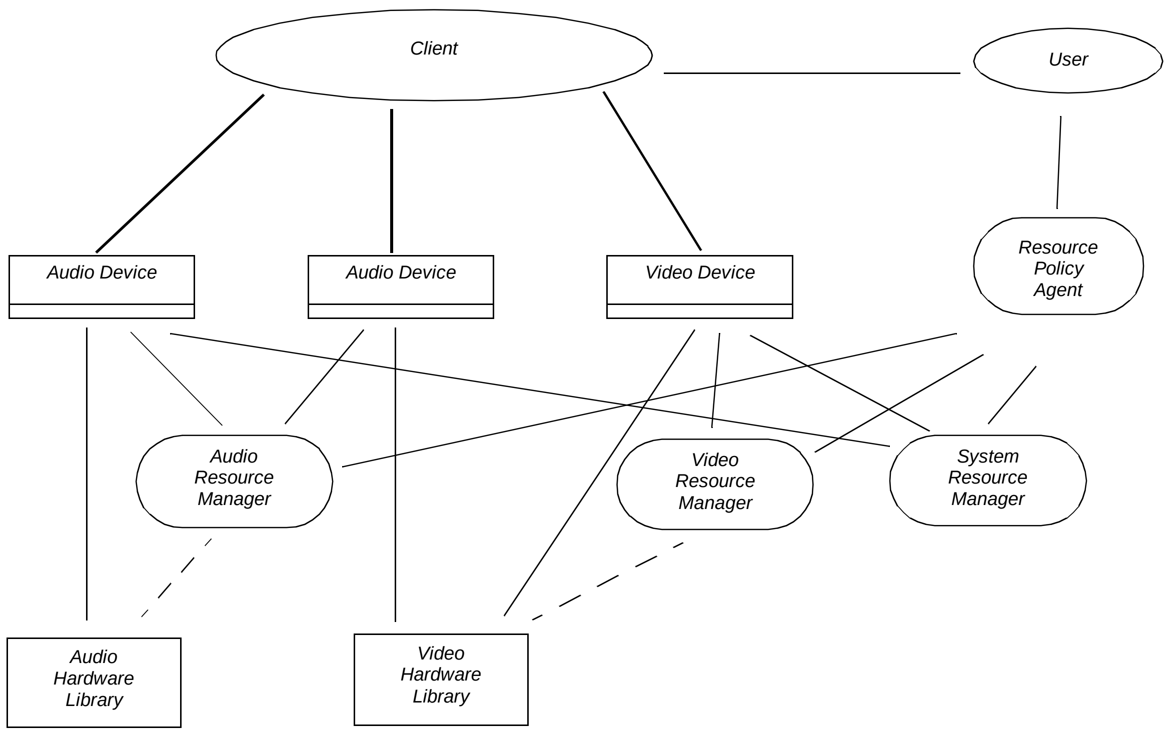


Figure 8 — A possible resource management scenario

9.1.4 Quality of Service Management

In order for virtual resources to be useful, physical resources have to be acquired. Because the available physical resources may vary considerably, and the variation may be dynamic, it is necessary to match the requirements of an application to the physical resources available, and to modify this matching of expectations dynamically.

This part of PREMO encapsulates this matching and modification process in *Quality of Service* (QoS) facilities. Hooks are provided for supporting QoS management, but no particular approach to QoS management or policy is imposed.

Four aspects of QoS may be distinguished.

a) expression of QoS requirements;
b) negotiation of an agreed QoS between provider and clients;
c) mechanism for reporting violations of the agreed QoS;
d) mechanism for adapting behaviour in response to QoS violations.

The expression of QoS requirements is supported by *QoSDescriptor* objects (see 7.4).

The setting of the configuration objects for a virtual device is done through separate operations. By doing so, it may happen that the requirements specified by these configuration objects cannot be satisfied, e.g., the quality of the service has to be degraded. The *VirtualResource* object defines an operation, called *validate*. This operation informs the invoker whether a given set of requirements can be satisfied, and if they cannot, then it attempts to suggest alternatives that can be satisfied.

Depending on the subtype of a virtual resource, QoS management can be attached to an individual port of a virtual device (see 9.2.2) or to the virtual resource as a whole.

Notification of QoS violations is supported by a callback mechanism, using PREMO events; if a *VirtualResource* identifies a QoS violation, it will send an event to a registered *Callback* object. This could be used to invoke a QoS manager, though QoS managers are not standardized by this part of PREMO. The format of QoS violation events is as follows:

— the name of the event is "*QoSViolation*";

— the event data should contain the key–value pairs whose requirement have been violated;

— if the QoS management is attached to a port of a virtual device (see 9.2.2), the key "*PortID*", with a value identifying the port, should be added to the event data.

Specific subtypes of *VirtualResource* objects may add additional event data to this event.

The behaviour of a resource can be modified in response to a QoS violation by changing the *QoSDescriptor* and reinvoking the negotiation process. This part of PREMO does not standardize specific mechanisms and object types for doing this.

NOTE — An example for a possible policy is as follows. When a resource manager detects that it cannot satisfy the quality of service requirements of its virtual resource clients, usually in response to a new request to share a physical resource, it may have to pre–empt a running virtual resource. When this happens, an event is sent to the virtual resource notifying it that it has lost access to the physical resource; the virtual resource, in turn, generates an event with the name *"Resource Lost"*, notifying an event client that it has been pre–empted. At a later time, if the resource manager detects that it can meet the needs of a pre–empted virtual resource, it may send an event to the virtual resource notifying it that it has gained access to the physical resource; the virtual resource, in turn, generates an event with the name *"Resource Acquired"*, notifying the client that it has regained use of its resources.

9.2 Virtual Devices

A *VirtualDevice* is a subtype of *VirtualResource* that abstracts interaction processing or presentation capabilities. These devices may be hardware devices, such as capture and display cards, complex media renderers (see, e.g., the subtypes of *VirtualDevice* defined in ISO/IEC 14478-4, part 4 of PREMO), or they may be software "devices" such as compressor/decompressors. A virtual device may represent a resource internal to a system, such as an audio capture device, or it may represent a resource external to a system, such as a CATV tuner.

The virtual device abstraction is designed to provide a common way for clients to use a wide variety of capabilities in many kinds of operating environments. Some of the key considerations that influenced the *VirtualDevice* interface design are:

a) *Resource management:* Almost all devices require some system resources, such as CPU and memory, to perform their function. Some require the use of specialized hardware, access to which needs to be managed. In order to permit sharing of resources among several simultaneously active clients, multiple virtual devices may use a single physical resource at the same time. However, the client is insulated from the details of resource sharing and can largely behave as if it has exclusive access to the resource.

b) *Stream position control:* Devices typically operate on a stream of media data. The client interacts with a virtual device's *StreamControl* objects (see clause 8) to determine stream position and to control various aspects of stream progress. The degree to which a device can report or control the position in the data stream is determined by the subtype of *StreamControl* type with which it can be associated.

c) *Device abstraction:* Physical devices can vary substantially from manufacturer to manufacturer. Not only are the set of functions often different, but the way in which they are combined and accessed is commonly different. Yet an application writer does not want to write specialized code to operate on each manufacturer's device: a set of common abstractions is needed.

d) *Media format abstraction:* The format of media data processed by devices also varies considerably. There are many different kinds of audio encodings, and even more video encodings. Some encodings are quite complex, requiring a plethora of details to be properly negotiated to interpret the data correctly. The abstraction of media data may also encompass for complex, time–based descriptions of multimedia presentation data, see, e.g., ISO/IEC 14478-4, part 4 of PREMO. The client is typically interested in few (if any) of these details, whereas the Multimedia Systems Services framework must necessarily be concerned with all of them.

e) Virtual devices can also be used to allow modelling, rendering, and interaction to be uniformly integrated into a network of objects for managing the production and utilization of multimedia data. See, e.g., ISO/IEC 14478-4, part 4 of PREMO for further examples of such devices.

The construction of the Multimedia Systems Services *VirtualDevice* type has been designed to address these considerations while providing an expressive, flexible framework into which a large spectrum of media devices may be cast.

The following subclauses describes the roles of the various components that make up a virtual device. (These components are also depicted on Figure 9; note that only the shaded boxes, and their client–visible interfaces, are standardized by PREMO) All of these elements work together to provide the client a useful abstraction of a physical device. Separating the various functions related to a virtual device into distinct types permits a wide variety of physical devices to be represented cleanly.

9.2.1 Processing element

The processing element is an abstract representation of the part of a virtual device that performs the operations which are abstracted by a particular *VirtualDevice* type. It is the functionality of this virtual device component that determines a particular virtual device's position in the *VirtualDevice* type hierarchy. From the point of view of PREMO, processing elements are purely conceptual entities and clients never directly interact with them. A particular virtual device may have a complex internal structure. Consequently, this part of ISO/IEC 14478 does not contain a detailed specification of their interfaces. However, this abstraction is useful in describing the externally visible behaviour of virtual device objects.

9.2.2 Ports

The processing element gets its input data from and sends its output data to ports. For example, an MPEG decompressor could be defined as having a single input port and two output ports (one audio and one video). From a client's perspective, ports are distinguished by an abstract non–object data type, identifying individual ports. The port itself is not normally accessed by the client. It exists to perform data movement operations needed the Multimedia Systems Services framework.

9.2.3 Streams

The client can focus all inquiry and control methods concerning data stream position at the *StreamControl* interface. The client can obtain a reference to the virtual device's overall *StreamControl* object by through the (inherited) *stream* attribute.

In addition to the overall *StreamControl* object, *StreamControl* objects may be available to interact with the stream position at individual ports. The *VirtualDevice* type provides a *getPortPortConfig* operation that can be used to get references to the *StreamControl* object associated with individual ports. The *StreamControl* objects associated with individual ports are subtypes of the overall *StreamControl* object associated with the virtual device itself.

The immediate type of the *StreamControl* objects associated to the port(s) of the virtual device shall be a subtype of the device's overall *StreamControl* object. Virtual device may differ from one another in the choice of *StreamControl* object which they associate either on an overall level and/or on a port–by–port basis.

NOTE — For example, a virtual device for an MPEG decoder may declare a subtype of *StreamControl*, say, *MStreamControl*, which is the immediate type of its overall *StreamControl*. This requirement says that all *StreamControl* objects, exported by the MPEG decoder, shall be a subtype of *MStreamControl*.

A particular virtual device may decide not to associate a *StreamControl* object to one of its ports, i.e., the object reference refers to *NULLObject*; the same is valid for the device's overall *StreamControl* object. Also, the same reference for a *StreamControl* object may be returned for multiple ports and for the overall *StreamControl*.

NOTE — For example, a virtual device, simply interpreting an audio file and sending the audio data on its output port, might be realized with a single output port defined for the device. In this case, the overall *StreamControl* and the *StreamControl* associated to that single output port might be the same, i.e., the object references would refer to the same object instance.

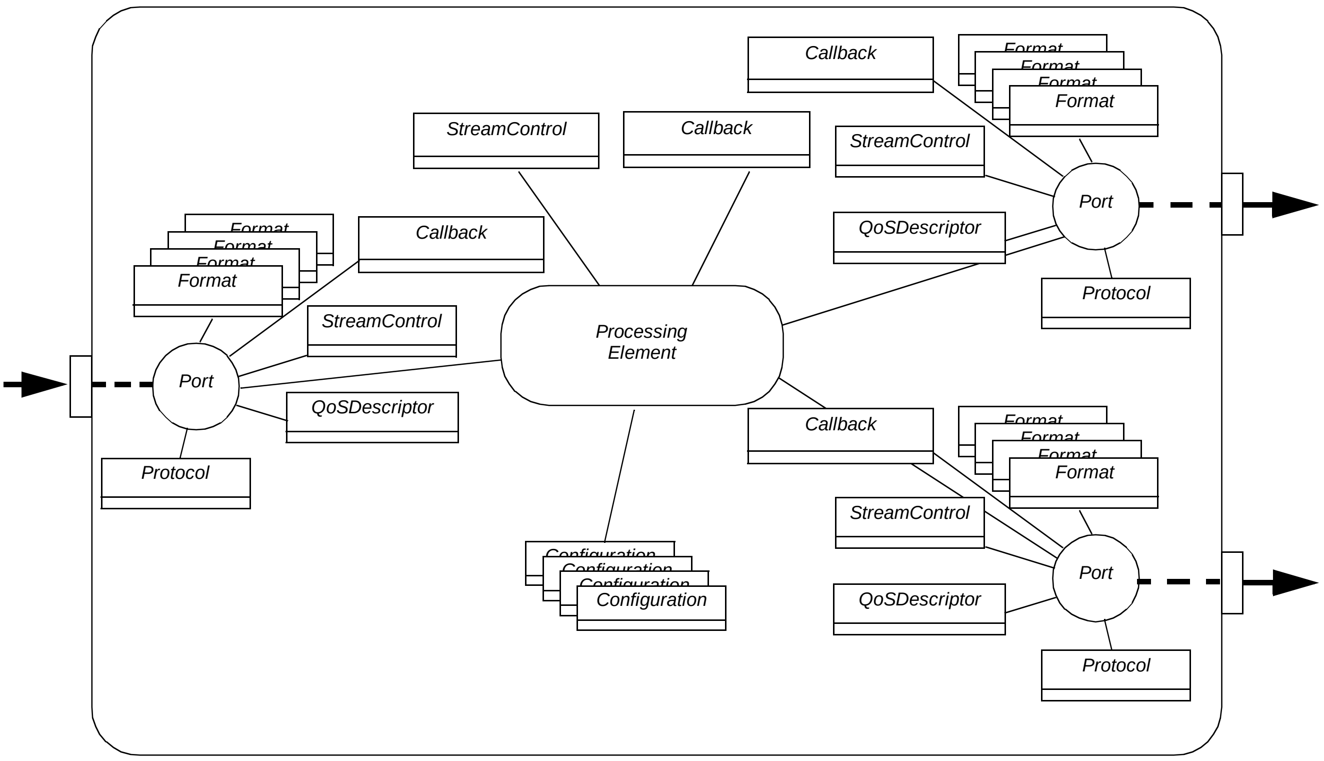


Figure 9 — Virtual Device type

9.2.4 Port configurations

As a subtype of *VirtualResource*, a virtual device object is described through a set of associated configuration objects. In the case of a virtual device, some of these configuration objects are grouped around ports. More specifically, each port has the following configuration objects associated to it.

a) A sequence of *Format* objects, ordered in time. Media information flowing through the port (and controlled by a *Stream-Control*) may change the associated format when a specific relative time value is reached (i.e., time is relative to the stream flow, see also 7.9.2 of ISO/IEC 14478-2). The choice of the specific subtypes of *Format* object may be the only visible difference among various virtual devices.

b) A *Protocol* object, describing the media–independent aspects of the communication through the port.

c) A *QoSDescriptor* object, setting quality of service requirement on the port.

d) A reference to a *Callback* object, which acts as a handler for port–specific events.

A special structure type, called *PortConfig*, is defined to describe port configurations (see 10.4.1). The *VirtualDevice* object provides the necessary operations to access these structures and to set and/or retrieve their structure tags.

The *VirtualDevice* object, as a subtype of *VirtualResource*, inherits the *validate* operation (see 9.1.4), which checks whether the requirements set through configuration objects can be met or not. Furthermore, the *VirtualDevice* object also defines the *portValidate* operation, which performs a similar check on the configuration assigned to a specific port.

Note that not necessarily all configuration objects, associated to a virtual device, are also directly associated to a port. Separate operations are defined to access the configuration objects which are defined on the virtual device level only.

9.3 Virtual Connections

The virtual connection is an object that represents the application's abstract view of media transport between virtual devices. A virtual connection does not actually transport the media; this functionality is the responsibility of the appropriate virtual device(s). A virtual connection is responsible for negotiating the connection "agreement" between the virtual devices and to provide a focal point for command and status information of the actual connection.

Description of a virtual connection object uses the concept of a *virtual connection adapter*; this is a framework internal concept that the virtual connection will instantiate to transport the media between the two virtual devices. PREMO does not standardize the exact behaviour of this entity. This adapter is required when the virtual devices are on separate systems or the virtual device ports are incompatible for direct media transfer. The virtual connection adapter is a private construct of the virtual connection and is not visible to the client.

9.3.1 Examples for connection agreement

As an example, the virtual connection in Multimedia Systems Services may constitute an "agreement" on the following:

a) *Media type to be transported between the two virtual device ports (including media master)*

Each port in a virtual device will have an associated *Format* object which defines what media types can be supported (i.e., produced or consumed). It is possible for the *Format* object to support multiple media types. For example, the *DigitalAudio-Format* supports ulaw, alaw, linear and ADPCM encodings. An agreement of the media type is reached when the *Format* object of the source virtual device and the *Format* object of the target virtual device have equal characteristic values. The *Format* object is a subtype of *EnhancedPREMOObject* and, as such, provides a *matchProperty* operation which can be used to compare the characteristic values of one *Format* object with another. For setting the characteristic value, the interface of *Format* provides the appropriate methods inherited from the interface of *PropertyConstraint* (see 9.12.2 of ISO/IEC 14478-2).

The setting of the characteristic values in the source and target *Format* objects can be accomplished in the following ways:

1) *Client media master*: The client sets the characteristic values of both *Format* objects. In this case, the virtual connection is only responsible for insuring that the two *Format* objects are compatible.

2) *Device media master*: The client sets one of the *Format* objects (the media master) and the virtual connection will negotiate the setting of the other *Format* object.

3) *Connection media master*: The client sets neither and the virtual connection will negotiate the settings of both *Format* objects by first querying their respective capabilities and then using the appropriate operations on both objects to find a match.

b) *Type of connection*

These connections will generally be of one of four types:

1) hardware,

2) direct,

3) local, or

4) network.

The virtual connection will determine the appropriate connection type.

Each port will identify its characteristics, some of which can be retrieved by the virtual connection accessing the port configuration structure and some of which must be retrieved by the virtual connection using a private port access (i.e., through means not standardized by PREMO). These characteristics may include such things as master/slave, PIO/DMA/shared memory/LAN/WAN, etc. The virtual connection will also determine if the virtual devices are in the same address space, in separate address spaces but on the same machine, or on separate systems. The virtual connection will use this information to determine the optimum type of connection which can be made. Examples of various connection settings can be found in Annex B.

A virtual connection adapter is required when a separate entity is needed to transport the data from one virtual device to another. This occurs in the following cases:

1) When the virtual devices are on separate machines.

5) When the virtual devices are on the same machine but can't agree on who drives the data movement.

6) When the virtual devices are on the same machine but in different processes with incompatible buffer managers.

In the case where the virtual connection adapter crosses system boundaries, the virtual connection adapter will be responsible for interfacing with the appropriate networking facilities to transport the data. This will require selecting a virtual connection adapter that is appropriate (network protocol wise) for both systems. An implementation of Multimedia Systems Services will offer several classes of virtual connection adapters in order to support a variety of networking protocols. Virtual connection adapters could be implemented to use any of the following network transports: TCP(UDP)/IP, NETBIOS, ATM, etc.

c) Quality of Service

The connection will also constitute an agreement on the quality of service. The *QoSDescriptor* parameters are an integral part of the virtual connection and are a reflection of the expectations of the application. This quality of service is directed partly by the *QoSDescriptor* object associated with the virtual connection, and partly by the *QoSDescriptor* objects associated with the ports of the connection. Based on the various quality of service property values at all these *QoSDescriptor* objects, the virtual connection object can establish the optimal quality of service it can honour. See 7.4 for a further explanation of the various quality of service parameters.

d) Stream & Synchronization capabilities

The parameters in this area of agreement describe the:

1) data exchange mechanism,

2) time, and

3) synchronization mechanisms and policies.

The virtual connection will determine if the virtual devices can agree on a common data exchange mechanism. The virtual connection will also determine the type of *StreamControl* object associated with each virtual device as well as with the connection ports. Using this information, the virtual connection will, if necessary, instantiate the appropriate virtual connection adapters.

9.3.2 Connection establishment

Once the virtual connection is created, the client can connect the two virtual devices by calling the *connect* operation of the *VirtualConnection*. Arguments of this operations are the references of the virtual devices and the respective ports on these devices:

connect

$deviceMaster_{in}$: *RefVirtualDevice*
$portMaster_{in}$: *Port*
$deviceSlave_{in}$: *RefVirtualDevice*
$portSlave_{in}$: *Port*

The order of the parameters does not signify the direction of the media flow (i.e. who is the source and who is the target). The arguments identify the media master, which allows the client to specify to the virtual connection which *Format* object should be used as the media master. Once the port characteristics have been obtained, the virtual connection will negotiate the format and the connection type. A virtual connection object can manage only one connection at a time.

A client can disconnect the virtual devices by invoking *VirtualConnection.disconnect* operation.

9.3.2.1 Unicast and multicast

The Multimedia Systems Services provides two types for virtual connections: unicast and multicast.

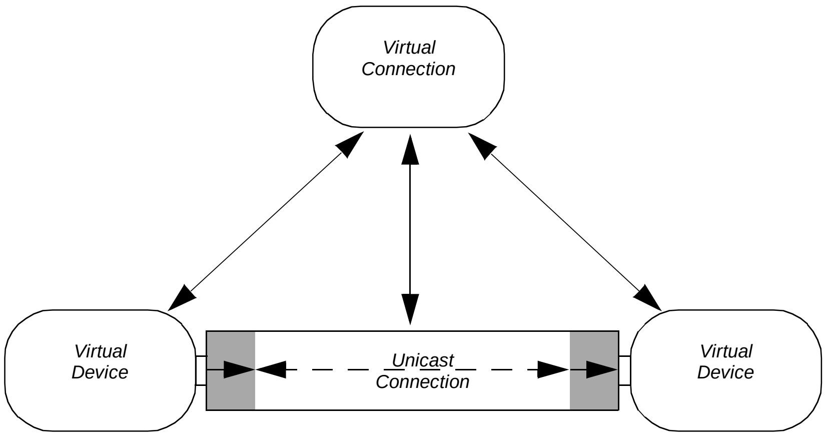


Figure 10 — Unicast connection

9.3.2.1.1 Unicast

The *VirtualConnection* is the base type for virtual connections and provides the support for unicast connections. It provides the base methods for creating and destroying a connection between two virtual devices, namely *connect* and *disconnect* as described above. An example of a unicast connection is shown in Figure 10.

9.3.2.1.2 Multicast

The Multimedia Systems Services also provides a *VirtualConnectionMulticast* type for supporting multicast connections. A multicast connection differs from the unicast connection in that the (master) output port provides a single instance of the data for all the connections, or a (master) input port may receive data from several connections. An example is shown in Figure 11.

The *VirtualConnectionMulticast* class provides the new operations to attach and detach new ports to connections. These operations provide support for dynamic multicast connections, that is, connections may be added or dropped as the stream is flowing.

9.4 Groups

It is often desirable to manipulate multiple resources as a group. When expressing quality of service requirements, for example, the client often cares more about such things as end–to–end delay of a set of connected devices rather than the delay of individual elements. When controlling the movement of data through the set of connected devices, it is often more convenient to manipulate a single object, rather than having to manipulate the objects associated with each of the resources.

To address these and other problems, the Multimedia Systems Services provides a *Group* object type to allow grouping of *VirtualResource* objects. This supports the grouping of *VirtualDevice*, *VirtualConnections*, and *Groups* (i.e., hierarchical *Groups* are allowed).

The *Group* is itself a subtype of *VirtualResource*. The additional operations it defines are to add/remove *VirtualResource* objects to/from a *Group*. A single *VirtualResource* can be added to or removed from a *Group* using the *addResource* and *removeResource* operations. In addition, it is possible to add or remove an entire graph of resources to a group using the following *addResourceGraph* and the *removeResourceGraph* operations. These operations add or remove a specified resource as well as all other virtual devices and virtual connections that are currently connected to it by a chain of device–connection data–flow pathways.

The role of the *Group* type is to perform the following operations:

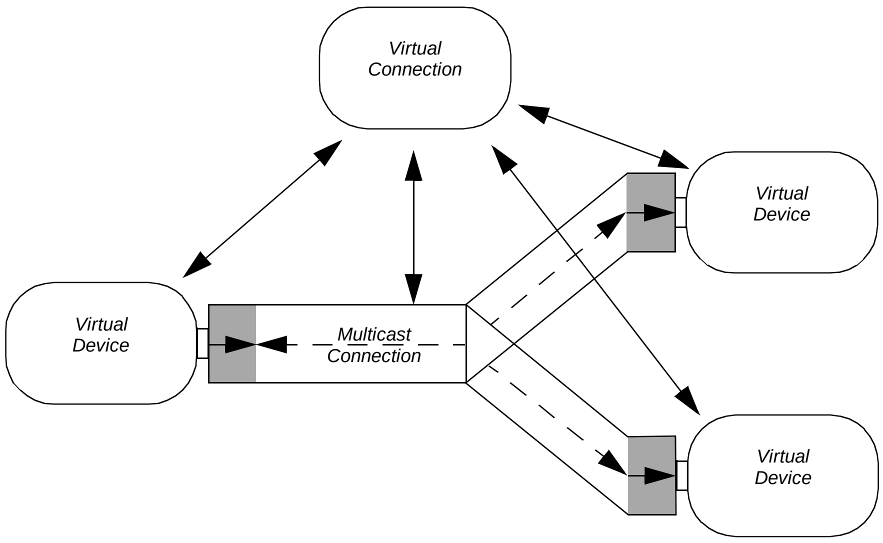


Figure 11 — Multicast connection

a) resource acquisition and end–to–end quality of service, and

b) stream control.

The definition of the *Group* in the Multimedia Systems Services specification does not provide support for automatic synchronization of a group of *VirtualResource* objects.

NOTE — This functionality may be added to the *Group* type in future revisions.

9.4.1 Resource acquisition and end–to–end QoS

In a situation where the client has created and connected several virtual resources to perform a specific function, the resource acquisition and quality of service specification procedures can be quite complex. While the ability to specify a (possibly different) *QoSDescriptor* for each virtual device and each virtual connection is a very powerful feature, it also places a heavy burden on the client to understand the intricacies of each and to specify the *QoSDescriptor* accordingly. In many situations, the client would like to specify only the end–to–end quality of service and have the system determine the necessary *QoSDescriptor* at each node. The ability to group a set of virtual resources and specify a single *QoSDescriptor* solves this problem for the client.

Once a set of virtual resources has been added to a group, the underlying physical resources required to realize the entire group can be obtained by calling the *VirtualResource.acquireResource* operation of the *Group*. The *QoSDescriptor* object assigned to the Group object (which can be accessed by the client through the *VirtualResource.resolve* operation) is taken to be the desired *QoSDescriptor* for the entire group. The group tracks the graph(s) that are contained within it, and the *QoSDescriptor* requirement is taken to be the end–to–end specification for all graphs. The group does the work of allocating the *QoSDescriptor* to individual objects to meet the overall *QoSDescriptor* objective.

Other problems can occur when a client executes the *VirtualResource.acquireResource* operation of each object independently. This can lead to resource deadlock or resource under–utilization. If the resources are acquired as a group, the system can help prevent these problems.

9.4.2 Stream control

Since the *Group* is a subtype of *VirtualResource*, it has an associated *StreamControl*. Formally, the *StreamControl* objects associated to a group are subtypes of the *StreamControl* object type in order to perform the special behaviour described below. These subtypes do not introduce any new operations, i.e., the interface of these subtypes are identical to the interfaces of the *StreamControl* object, and PREMO does not formally introduce new types for these.

The *StreamControl* object associated to a group provides the client progress and control information on the flow of data within the group. Once all the members of a group have been defined, the client can get the overall *StreamControl* object for the group using the *VirtualResource.streamControl* attribute.

The *StreamControl* objects of the constituent resources can be thought of as children of the (parent) group's *StreamControl* object. Operations invoked on the group object's *StreamControl* are forwarded as appropriate to its children: calling *pause* on a *Group*'s *ControlledStream* object causes all of its children to be paused, the same holds for operations like *prime*, *drain*, etc. This allows the client to call a single stream method, and have this command propagated to each stream of the members of the group. How exactly the control over the children's *StreamControl* objects is done is implementation dependent.

Some of the operations, like *prime* or *drain*, may also generate private control flow among the streams controlled by the group. E.g., if a *prime* request is forwarded by the group to a *StreamControl* associated to an input port, the implementation may also generate a control information to the *StreamControl* object "up–stream", i.e., the stream providing the data. Whether such additional protocol is defined or not is implementation dependent, and is not visible to the client of the group.

NOTE — An example for a *Group* might be an object whose children are a video and an audio device, with the audio device playing the sound track of the video. Using the group, a client may perceive a single media only, with video and audio synchronized; control over this media can be done by controlling the *StreamControl* object of the group. How such a control is mapped against the control of the separate video and audio is invisible to the client.

Methods invoked on the child *StreamControl* objects are not forwarded to the parent object; however, the parent *StreamControl* object or the state of other children's *StreamControl* objects might change as a side effect. For example, pausing a child's *StreamControl* object would not pause parent or sibling *StreamControl* objects, but these objects might enter some kind of stalled state as a result of flow control.

9.5 Logical Devices

The *LogicalDevice* object type is defined to provide a mechanism for hierarchically constructing larger, reusable device types from the set of devices available to an application

A logical device is a subtype of both a *Group* and a *VirtualDevice*. As a group, the operations needed to add a device to the logical device are already available, and it can also control the media stream among the devices within the group.

A logical device initially contains *no* ports in its interface. Conceptually, the client has to make the port of a device *contained within the logical device* visible to the outside through an explicit action which associates this port to a newly created port on the logical device itself. Formally, this is done through the *definePort* operation that takes a reference to a device already contained within the logical device, plus a port (which must be valid for the device) and returns the identifier of a new port that is added to the logical device interface and connected to the specified port of the original device. The port configuration of the newly created port will be identical to the configuration of the port it has been associated with.

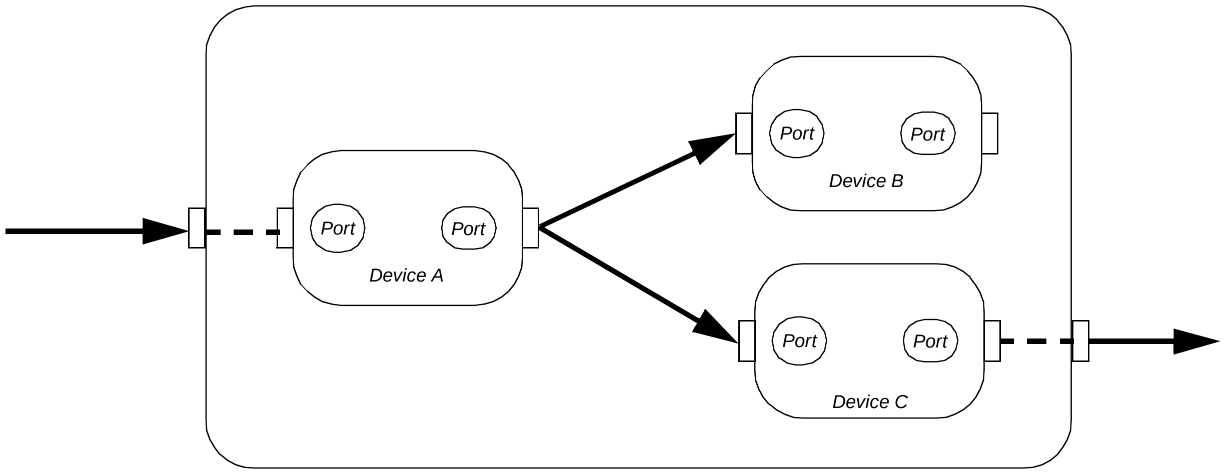


Figure 12 — Usage of logical device

Once the identification of ports is done, the logical device instance may be incorporated into a network of other virtual devices, and possibly added to another logical device instance.

NOTE — Figure 12 shows the possible usage of a logical device. The device contains three logical devices, denoted by device *A*, *B*, and *C*. The input port of device *A* is made visible to the outside, and so is the output port of device *C*. However, the ports of device *B*, as well as the other ports of devices *A* and *C*, are not made visible. To the outside world, the logical device behaves as a *VirtualDevice* with two ports (one input and one output), and the configuration of these ports are identical to the configuration of the relevant ports of the devices *A* and *C*.

10 Functional specification

10.1 Introduction

This clause provides the detailed functional specification of the non–object types and the object types that together define the PREMO Multimedia Systems Services component. The notation used in this clause follows the rules detailed in Annex A of ISO/IEC 14478-1.

Additionally to the object type definition, each MSS type may have a predefined set of properties and/or capabilities. These are defined in separate tables following the type specification schema. These tables include the name of the key, the type of the value, a flag whether the property is read only (R.O.) or not (R/W), and possibly a short description of the property. Capabilities are defined in a separate table. Subtypes may extend the values of capabilities.

10.2 Non–object data types

This sub–clause also defines all data types used by operations defined on PREMO MSS object types.

Additional state constants for streams:

$$MUTED : State \mid MUTED = 4$$
$$PRIMING : State \mid PRIMING = 5$$
$$DRAINING : State \mid DRAINING = 6$$

Identification of ports within virtual devices:

$Port == \mathbb{Z}$

Port types:

$PortType == INPUT \mid OUTPUT$

Semantic names of configuration objects in virtual resources:

$SemName == String$

Configuration information tuple, used to identify configuration objects virtual devices:

$ConfInfo == SemName \times ObjectType$

Indication of whether resources are acquired in a virtual resource or not:

$ResourceState == ACQUIRED \mid RELEASED$

10.3 Exceptions

This sub–clause lists the exception raised by operations defined in PREMO MSS types and which are not defined as part of the exceptions defined for ISO/IEC 14478-2 (see 9.3 of ISO/IEC 14478-2).

The list of the exceptions raised by operations defined on PREMO foundation object types are as follows.

ConfigurationMismatch	==	*Exception*
InvalidAccess	==	*Exception*
InvalidDevice	==	*Exception*
InvalidName	==	*Exception*
InvalidPosition	==	*Exception*
InvalidPort	==	*Exception*
PortMismatch	==	*Exception*
ResourceNotAvailable	==	*Exception*

10.4 Structures

10.4.1 Port information structure

This structure is used to describe a port configuration.

PortConfig

SimplePREMOObject

eventHandler: RefCallback
streamControl: RefStreamControl
qos: ConfInfo
protocol: ConfInfo
formats: seq (*Time* × *ConfInfo*)

The object types in the tuples refer to subtypes of *QoSDescriptor*, *MultimediaStreamProtocol*, and *Format* object types, respectively.

PortConfig

10.5 Configuration object

10.5.1 Format objects

10.5.1.1 *Format* object

Format$_{abstract}$

PropertyConstraint

Format

Properties defined:

Key	Type of Value	R.O or R/W	Description
NameK	*SemName*	R.O.	Semantic name of the format object

Capabilities defined:

None.

10.5.2 Transport and Multimedia Stream Protocol objects

10.5.2.1 *MultimediaStreamProtocol* objects

MultimediaStreamProtocol$_{abstract}$

PropertyConstraint

MultimediaStreamProtocol

Properties defined:

Key	Type of Value	R.O or R/W	Description
NameK	*String*	R.O.	Semantic name of the protocol object
VersionNumberK	$\mathbb{Z}$	R.O.	Implementation dependent value.
ByteOrderK	*String*	R/W	

Capabilities defined:

Key	Type of Value	Values
ByteOrderCK	array$_2$ *String*	<*"LittleEndian", "BigEndian"*>

10.5.2.2 *IntraNodeTransport* objects

IntraNodeTransport

MultimediaStreamProtocol

IntraNodeTransport

10.5.2.3 *InterNodeTransport* objects

InterNodeTransport

MultimediaStreamProtocol

InterNodeTransport

Capabilities defined:

Key	Type of Value	Values
NameCK	seq *String*	<*"TCP", "UDP", "RTP", "ATM", "NETBIOS"*>

10.5.3 Quality of Service objects

QoSDescriptor

PropertyConstraint

QoSDescriptor

Properties defined:

Key	**Type of Value**	**R.O or R/W**	**Description**
NameK	*String*	R.O.	Semantic name of the QoS descriptor object.
GuaranteedLevelK	*String*	R/W	
ReliableK	*Boolean*	R/W	Delivery of data is reliable or not. The type of the native property value is seq *Boolean*.
DelayBoundsK	$\mathbf{Z} \times \mathbf{Z}$	R/W	Minimum and maximum delay. The type of the native property value is $(\mathbf{Z} \times \mathbf{Z}) \times (\mathbf{Z} \times \mathbf{Z})$
JitterBoundsK	$\mathbf{Z} \times \mathbf{Z}$	R/W	Minimum and maximum jitter (delay variance). The type of the native property value is $(\mathbf{Z} \times \mathbf{Z}) \times (\mathbf{Z} \times \mathbf{Z})$
BandwidthBoundsK	$\mathbf{Z} \times \mathbf{Z}$	R/W	Minimum and maximum bandwidth. The type of the native property value is $(\mathbf{Z} \times \mathbf{Z}) \times (\mathbf{Z} \times \mathbf{Z})$

Capabilities defined:

Key	**Type of Value**	**Values**
GuaranteedLevelCK	seq *String*	<*"Guaranteed"*, *"Best Effort"*, *"NoGuarantee"*>
DelayBoundsCK	$(\mathbf{Z} \times \mathbf{Z}) \times (\mathbf{Z} \times \mathbf{Z})$	Delay bounds (range of minimum and range of maximum values).
JitterBoundsCK	$(\mathbf{Z} \times \mathbf{Z}) \times (\mathbf{Z} \times \mathbf{Z})$	Jitter bounds (range of minimum and range of maximum values).
BandwidthBoundsCK	$(\mathbf{Z} \times \mathbf{Z}) \times (\mathbf{Z} \times \mathbf{Z})$	Bandwidth bounds (range of minimum and range of maximum values).
MutablePropertyListCK	seq *Key*	<*"GuaranteedLevelK"*, *"ReliableK"*>
DynamicPropertyListCK	seq *Key*	<*"DelayBoundsK"*, *"JitterBoundsK"*, *"BandwidthBoundsK"*>

10.6 Stream Controls

10.6.1 *StreamControl* object

The *StreamControl* object is described finite state machine. The state transition table below is an extension of the transition table for *Synchronizable* objects (also valid for *TimeSynchronizable* objects) in 9.11.1 of ISO/IEC 14478-2.

To: / *From:*	STOPPED	STARTED	PAUSED	WAITING	MUTED	PRIMING	DRAINING
STOPPED	Y	Y	N	N	Y	Y	N
STARTED	Y	Y	Y	I	Y	Y	Y
PAUSED	Y	Y	Y	N	Y	Y	Y
WAITING	Y	Y	Y	N	Y	Y	Y
MUTED	Y	Y	Y	I	Y	Y	Y
PRIMING	Y	Y	I	I	Y	Y	Y
DRAINING	Y	Y	I	N	N	Y	Y

StreamControl[C]

PropertyInquiry
TimeSynchronizable[*C*] redef *(start, stop, pause, resume)*

start	==	σ*(STARTED, STOPPED* | *STARTED)*
stop	==	σ*(STOPPED)*
pause	==	σ*(STOPPED, STOPPED)* ⊕ σ*(PAUSED)*
resume	==	σ*(STARTED, STARTED* | *MUTED* | *PRIMING* | *DRAINING)* ⊕ σ*(STARTED* | *MUTED* | *PRIMING, WAITING)* ⊕ σ*(STARTED* | *MUTED* | *PRIMING* | *DRAINING* | *PAUSED, PAUSED)*
mute	==	σ*(MUTED, STOPPED* | *STARTED* | *MUTED)*
prime	==	σ*(PRIMING)*
drain	==	σ*(STOPPED, STOPPED)* ⊕ σ*(DRAINING)*

The effect of the *resume* operation when in *WAITING* or *PAUSED* states is to restore the state of the object to what it was *before* it arrived to *WAITING* or *PAUSED* state, *except* if the transition was internal to the object from *PRIMING* or *DRAINING* states to *PAUSED*. In the latter case, resume does not change the state.

Exceptions raised:

WrongState	The state transition operation is issued in an unallowed state.

StreamControl

10.6.2 ***SyncStreamControl* object**

SyncStreamControl[C]

TimeSlave[*C*]
StreamControl

SyncStreamControl

10.7 Devices, resources

10.7.1 *VirtualResource* object

VirtualResource$_{abstract}$

PropertyInquiry

resourceEventHandler: RefCallback
streamControl: RefStreamControl [Retrieve Only]

The virtual device may act as an event source, through *resourceEventHandler*, for resource management events. If the value of *resourceEventHandler* is *NULLObject*, all events are ignored. The initial value of the attribute is *NULLObject*. The possible event names are *"Resource Lost"* and *"Resource Acquired"*. See 9.1 for further details.
streamControl identifies the *StreamControl* object instance associated to the virtual resource. Its value may be *NullObject*.

resourceState: ResourceState [Retrieve Only]

resourceState indicates whether the resources are currently acquired or not.

resolve

semanticName$_{in}$*: SemName*
objRef$_{out}$*: RefPropertyConstraint*
exceptions: {*InvalidName*}

The operation returns the reference to its associated configuration object whose semantic name is *semantic-Name*$_{in}$.

Exceptions raised:

InvalidName	No configuration object with name *semanticName*$_{in}$ is associated to this object.

acquireResource

exceptions: {*ResourceNotAvailable*}

The resource(s), managed by the virtual resource object, are acquired. In the case the resource can be allocated, the *bind* operation for all associated configuration objects is also invoked.

Exceptions raised:

ResourceNotAvailable	The required resource could not be allocated.

releaseResource

The resource(s) managed by the object are released. The *unbind* operation for all associated configuration objects is also invoked.

Exceptions raised: None.

validate

$valid_{out}$: *Boolean*
$validTuples_{out}$: seq(*SemName x* seq(*Key x* seq *Value*))

The object validates whether the configuration requirements, associated with the *VirtualResource,* can be satisfied or not. The result is returned in $valid_{out}$. In the case where this return value is *FALSE*, the object attempts to propose a replacement for the configuration objects, i.e., a sequence of key–value pairs for a sequence of configuration objects. This proposed replacement attempts to satisfy the original settings as far as this is possible. The client is responsible to set these values, if accepted. If no alternative is possible, the names are empty.

Exceptions raised: None

VirtualResource

Properties defined:

Key	Type of Value	R.O or R/W	Description
ConfigurationNamesK	seq *ConfInfo*	R.O.	Semantic names and types of all associated configuration objects. The type of the native property value is analogous to the property type.

Capabilities defined:

None.

10.7.2 *VirtualDevice* object

VirtualDevice$_{abstract}$

VirtualResource

resourceEventHandler: RefEventHandler
ports: seq *Port* [Retrieve Only]
streamControl: RefStreamControl [Retrieve Only]
configurations: seq *ConfInfo*

This *resourceEventHandler* attribute is inherited from *VirtualResource*. Additionally to the possible event names specified for *VirtualResource*, a virtual device may also raise events with the names:
"State Changed Event" and *"Format Changed Event"*.
ports list all port identifiers which are available on the device. *streamControl* is inherited from *VirtualResource*; it identifies the stream control object which is not associated to any port. *configurations* identifies the configuration objects which are not associated to any port.

getConnection

port$_{in}$*: Port*
connection$_{out}$*: RefVirtualConnection*
exceptions: {*InvalidPort*}

A reference to the *VirtualConnection* object, associated to a port, is returned. This information may be provided to the device by a virtual connection through an operation which is not visible to the client.

Exceptions raised:

InvalidPort	*port*$_{in}$ does not identify a port on that object.

portValidate

port$_{in}$*: Port*
formatName$_{in}$*: String*
valid$_{out}$*: Boolean*
validTuples$_{out}$*:* seq(*SemName x* seq(*Key x* seq *Value*))
exceptions: {*InvalidName, InvalidPort*}

The object validates whether the protocol and quality of service requirements, associated with the port identified by *port*$_{in}$, can be satisfied with the format identified by *formatName*$_{in}$. The result is returned in *valid*$_{out}$. In the case where this return value is *FALSE*, the object attempts to propose an optimal replacement for the associated protocol and quality of service descriptor objects, based on the global configuration objects associated to the virtual device object and based on the previously defined values for the port configuration objects. If accepted, the client is responsible to set these values. If no alternative is possible, the return sequence is empty.

Exceptions raised:

InvalidName	No configuration object, with name *formatName*$_{in}$ is associated to this object
InvalidPort	*port*$_{in}$ does not identify a port on that object.

getPortConfig

portId$_{in}$*: Port*
portConfig$_{out}$*: RefPortConfig* [Shallow Copy]
portType$_{out}$*: PortType*
exceptions: {InvalidPort}

The operation returns a port configuration structure for the port identified by *portId*$_{in}$.

Exceptions raised: None.

setPortConfig

port$_{in}$*: Port*
portConfig$_{in}$*: RefPortConfig* [Shallow Copy]
exceptions: {*InvalidName*, *InvalidPort*, *InvalidPosition*}

The operation replaces (if no exceptions are raised) the port configuration associated to port *porti*$_n$ by the structure *portConfig*$_{in}$.

Exceptions raised:

InvalidName	No configuration object, with name *semanticName*$_{in}$ is associated to this object
InvalidPort	*port*$_{in}$ does not identify a port on that object.
InvalidPosition	A position value, associated to a format, is invalid.

VirtualDevice

Properties defined:

Key	**Type of Value**	**R.O or R/W**	**Description**
InputPortK	$\mathbb{Z}$	R.O.	Number of input ports on the device.
OutputPortK	$\mathbb{Z}$	R.O.	Number of output ports on the device.
InputFormatTypesK	seq*(Port* × *seq ConfInfo)*	R.O.	Types of *Format* objects which can be associated to specific input ports.
OutputFormatTypesK	seq*(Port* × *seq ConfInfo)*	R.O.	Types of *Format* objects which can be associated to specific output ports.
GlobalFormatTypesK	seq *ConfInfo*	R.O.	Types of *Configuration* objects which can be associated to the global configuration objects (i.e., which are not associated to any port)

Capabilities defined:

Key	**Type of Value**	**Values**[a]
InputPortCK	$\mathbb{Z}$	*maximum*
OutputPortCK	$\mathbb{Z}$	*maximum*
InputFormatTypesCK	seq *ObjectType*	Allowed input format types in object
OutpuFormatTypesCK	seq *ObjectType*	Allowed input format types in object
GlobalFormatTypesCK	seq *ObjectType*	*<QoSDescriptor, Format>*

a. The values for the capabilities are finalized in the specification of the various subtypes of *VirtualDevice.*

10.7.3 Virtual connections

10.7.3.1 *VirtualConnection* object

VirtualConnection$_{abstract}$

VirtualResource redef (*acquireResource*)

connect

deviceMaster$_{in}$*: RefVirtualDevice*
portMaster$_{in}$*: Port*
deviceSlave$_{in}$*: RefVirtualDevice*
portSlave$_{in}$*: Port*
exceptions: {ConfigurationMismatch, InvalidPort, PortMismatch, ResourceNotAvailable}

The operation connects two virtual device ports. The port details negotiation starts with the port specified with *deviceMaster*$_{in}$ and *portMaster*$_{in}$. The input and output port is determined by the type of port connected.

Exceptions raised:

ConfigurationMismatch	The configurations cannot be reconciled. The exception data lists the conflicting configuration objects. It contains a sequence of tuples, each tuple being a pair of *ConfInfo* tuples (i.e., semantic name and object type). The first element of the tuple refers to the configuration object on the master, and the second on the slave.
InvalidPort	The port is invalid on the virtual device.
PortMismatch	Both ports are either output or input.
ResourceNotAvailable	The connection could not be established, e.g., because the object already manages a connection, or because some other resource is not available.

disconnect

The connections are broken. The operation does an implicit release.

Exceptions raised: None.

getEndpointInfoList

info$_{out}$: seq *(RefVirtualDevice* × *Port* × *Boolean)*

An information is returned on all connected ports: reference to the virtual device, its port identification, and a flag to identify whether the port is a master port (value *TRUE*) or not (value *FALSE*).

Exceptions raised: None.

VirtualConnection

Properties defined:

Key	Type of Value	R.O or R/W	Description
TransportTypeK	seq *ObjectType*	R/W	Protocol object used by the object instance.

Capabilities defined:

Key	Type of Value	Values
TransportTypeCK	seq *String*	*<TCPTransport, UDPTransport, RTPTransport, DMATransport, NETBIOSTransport, ATMTransport>*

10.7.3.2 *VirtualConnectionMulticast* object

VirtualConnectionMulticast$_{abstract}$

VirtualConnection

attach

device$_{in}$: RefVirtualDevice
port$_{in}$: Port
exceptions: {ConfigurationMismatch, InvalidPort, PortMismatch, ResourceNotAvailable}

The operation attaches a new slave port to the connection. This operation can be used once the initial connection is established using *VirtualConnection.connect*.

Exceptions raised:

ConfigurationMismatch	The configurations cannot be reconciled. The exception data lists the conflicting configuration objects. It contains a sequence of tuples, each tuple being a pair of *ConfInfo* tuples (i.e., semantic name and object type). The first element of the tuple refers to the configuration object on the master, and the second on the slave.
InvalidPort	The port is invalid on the virtual device.
PortMismatch	The new port is of an incompatible type (should be either input if the master is of output type, or output if the master is of input type).
ResourceNotAvailable	The connection could not be established.

detach

device$_{in}$: RefVirtualDevice
port$_{in}$: Port
exceptions: {PortMismatch}

The operation detaches a port from the multicast. Detaching the source port tears down the connection.

Exceptions raised:

PortMismatch	The port was not attached to the connection.

VirtualConnectionMulticast

10.7.4 *Group* object

$Group_{abstract}$

VirtualResource

addResource

$resource_{in}$: RefVirtualResource

The operation adds a new resource to the group.

Exceptions raised:
None.

removeResource

$resource_{in}$: RefVirtualResource
exceptions: {ResourceNotAvailable}

The operation removes the resource from the group.

Exceptions raised:

ResourceNotAvailable	The resource is not part of the group.

addResourceGraph

$resource_{in}$: RefVirtualResource

The operation adds a new resource and all resources to which it is connected to the group.

Exceptions raised: None.

removeResourceGraph

$resource_{in}$: RefVirtualResource
exceptions: {ResourceNotAvailable}

The operation removes the resource and all resources to which it is connected from the group.

Exceptions raised:

ResourceNotAvailable	The resource is not part of the group.

getResourceList

$resources_{out}$: seq *RefVirtualResource*

The operation returns all resources connected to the group.

Exceptions raised: None..

Group

10.7.5 *LogicalDevice* object

$LogicalDevice_{abstract}$

VirtualDevice
Group

definePort

$resource_{in}$: *RefVirtualDevice*
$port_{in}$: *Port*
$newport_{out}$: *Port*
exceptions: {*InvalidPort, InvalidDevice*}

The input to this operation is a reference to a virtual device and a port. The designated port is added to the interface of the device. The value of $newPort_{out}$ shall be used to identify the newly created port in subsequent operation calls involving port identifiers.

Exceptions raised:

InvalidDevice	The designated device is not contained in the logical device
InvalidPort	The resource is part of the device, but does not provide a port with the specified name.

LogicalDevice

11 Component specification

The basic profile of MSS contains the media independent types, i.e., the fundamental building blocks MSS defines to achieve interoperability. A number of applications, as well as further PREMO components, may rely on this profile only to define their own, media-specific functionality. The various specific devices defined by MSS are grouped into a separate profile.

MSSComponent

Basic

provides service

QoSDescriptor, IntraNodeTransport, InterNodeTransport, Format,
StreamControl, SyncStreamControl,
Group

provides type

PortConfig, ImageAOI,
QoSDescriptor, MultimediaStreamProtocol, IntraNodeTransport, InterNodeTransport, Format,
StreamControl, SyncStreamControl,
VirtualResource,
VirtualDevice, VirtualConnection, VirtualConnectionMulticast, Group, LogicalDevice

requires service

Component FoundationComponent Profile Extended

requires type

Component FoundationComponent Profile Extended

MSSComponent

Annex A
(normative)
Overview of PREMO MSS objects

This annex gives an overview of all PREMO Object types defined in this part. This Annex does not add any new information, and is here for easier reference only.

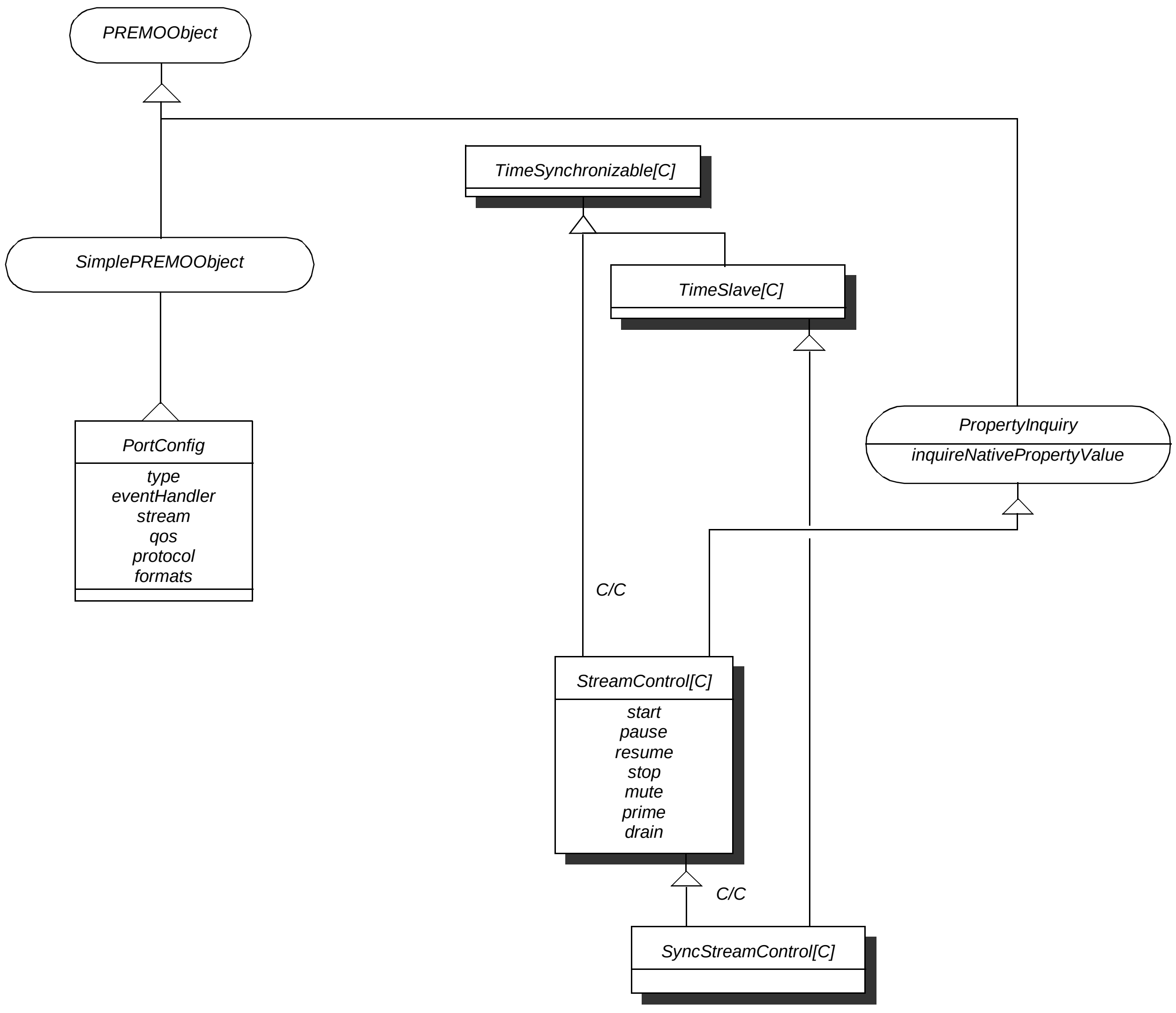


Figure 13 — PREMO MSS object types (structures, stream control)

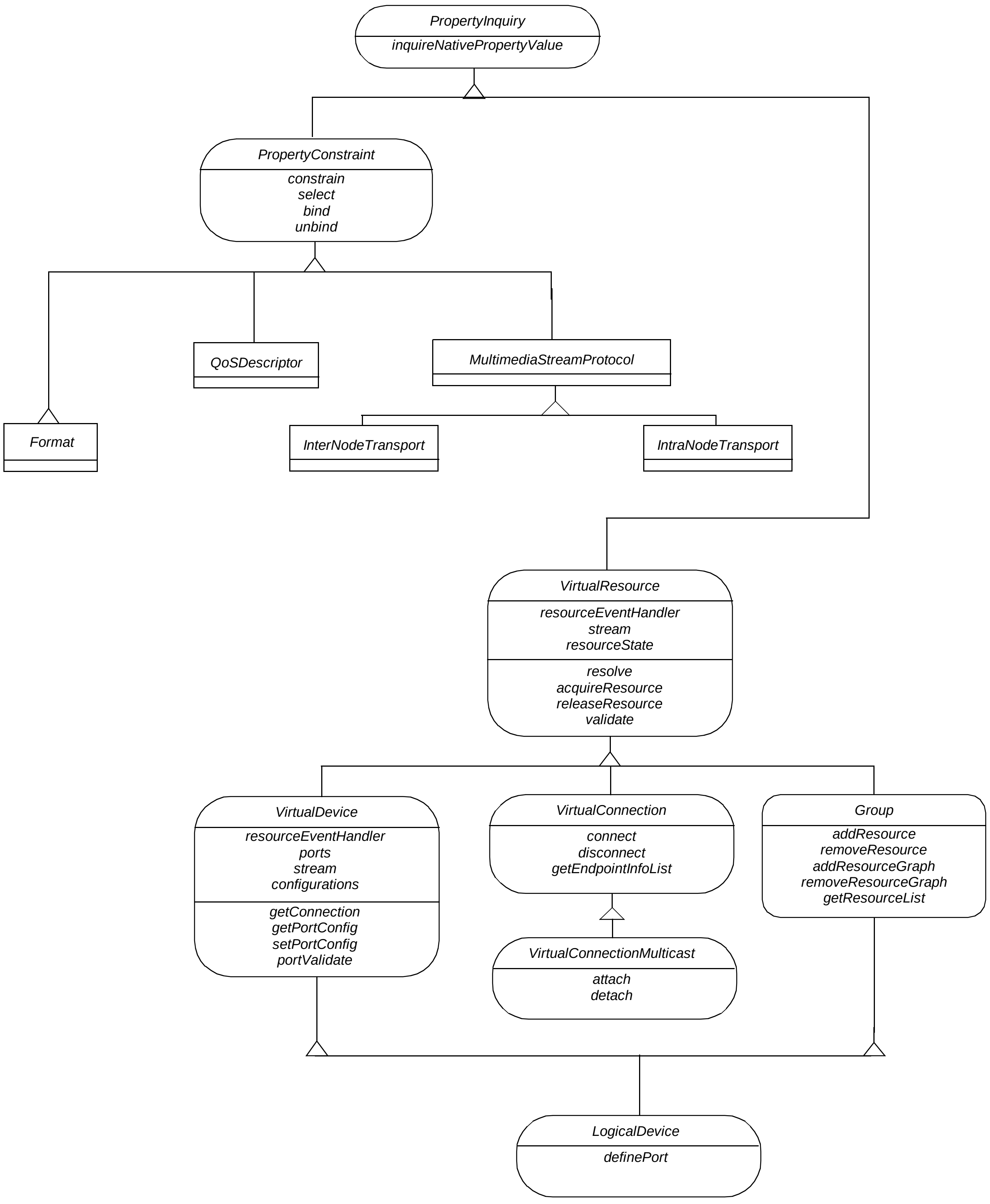
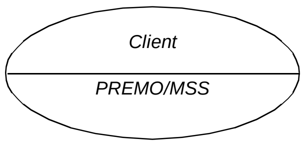


Figure 14 — PREMO MSS object types, cont. (configuration objects, virtual resources)

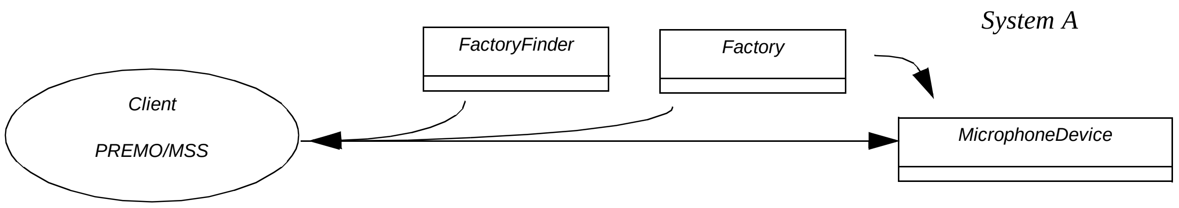
Annex B
(informative)
A typical example scenario for MSS usage

Here the actions of a client using the Multimedia Systems Services to perform a simple distributed multimedia action — capturing audio with a microphone on one system and playing through a speaker on another system — are traced, in a simplified form.

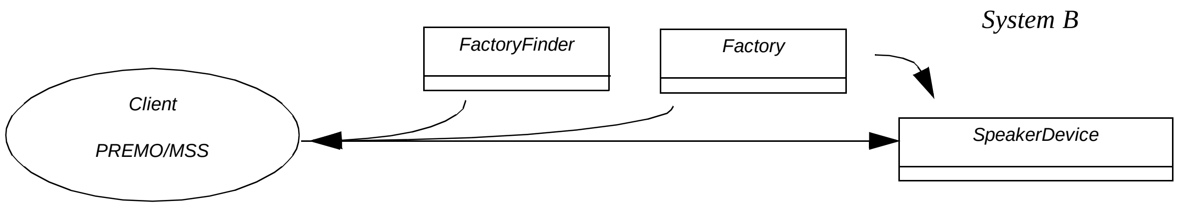
a) First, the client declares and initializes the PREMO Multimedia Systems Services client–side environment.



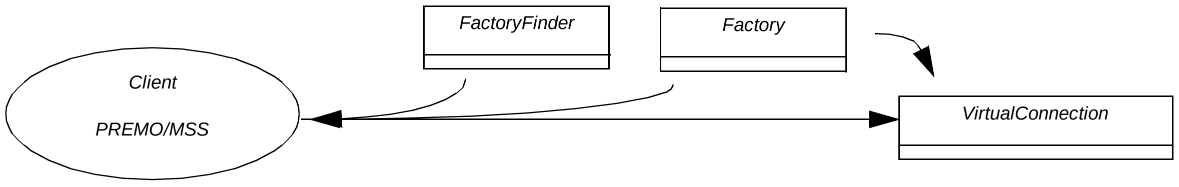
b) Now, the client builds a location–based capability for a microphone, and creates an instance of *MicrophoneDevice* on the correct system, using a *FactoryFinder* and a *Factory* instance. The location of the microphone is expressed in the property constraint passed to the *Factory*.



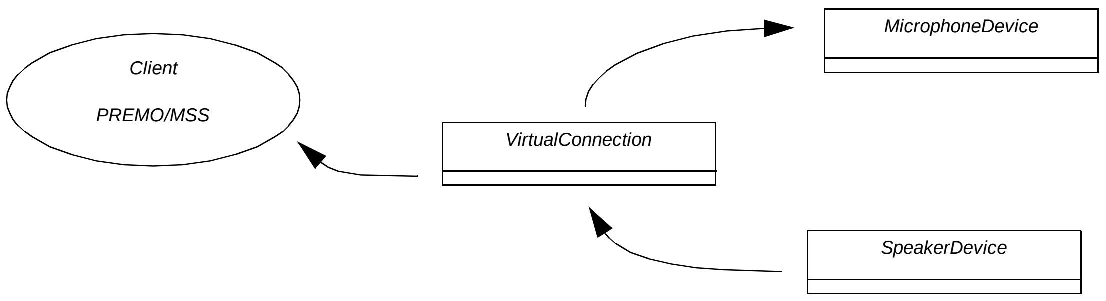
c) The client follows a similar process to create a *SpeakerDevice* on a different system.



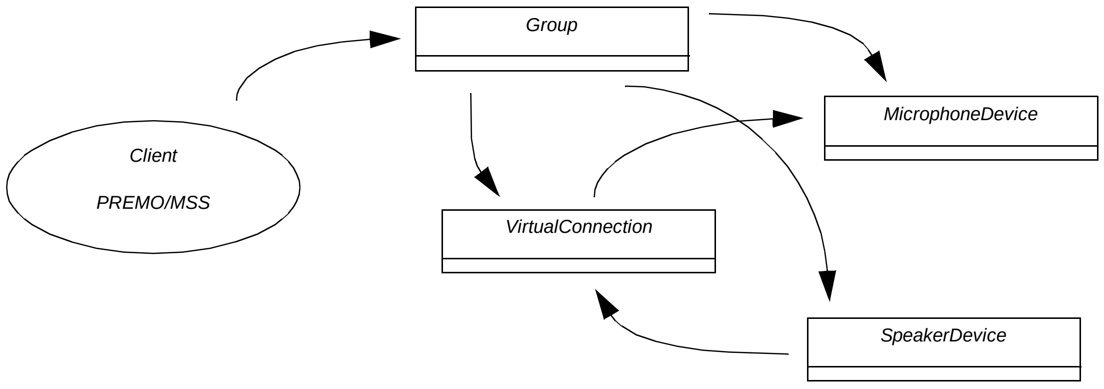
d) The client creates a *VirtualConnection* capable of connecting the two *VirtualDevices* on the two separate machines.



e) The client connects the two devices by sending a *VirtualConnection.connect* request.

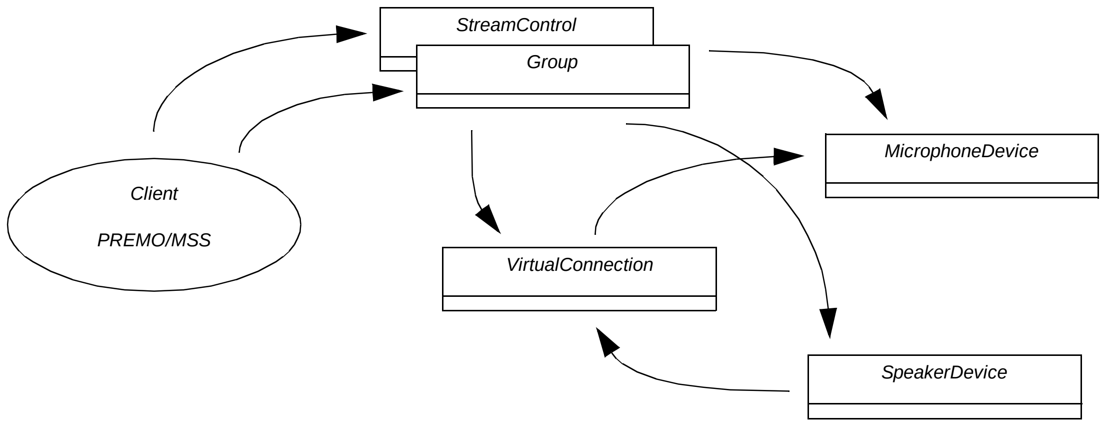


f) The client creates a *Group*, then adds all *VirtualResources* in the graph to the *Group* by sending the *Group.addResource-Graph* request.



g) The client causes the *Group* to acquire resources by sending a *VirtualResource.acquireResource* request, to obtain, for example, a reliable connection.

h) The client gets the *StreamControl* object for the *Group* by through the a *VirtualResource.streamControl* attribute and then starts the media stream.



Annex C
(informative)
Basic Devices

This annex describes a full hierarchy of format and virtual device objects which describe a set of basic multimedia devices. The subtypes defined in this annex do not intend to provide an exhaustive set of all possibilities to define, e.g., video access. The object types defined in this annex are of course usable as they are, but they should rather be considered as general patterns. More specialized specifications may be defined by other standardization bodies and various vendors to encompass particular needs. Especially, other components of PREMO may define further subtypes for some special use, e.g., computer graphics rendering.

A quick overview of the types defined in this Annex is given in Figure 15. The types in the diagram will be discussed in detail

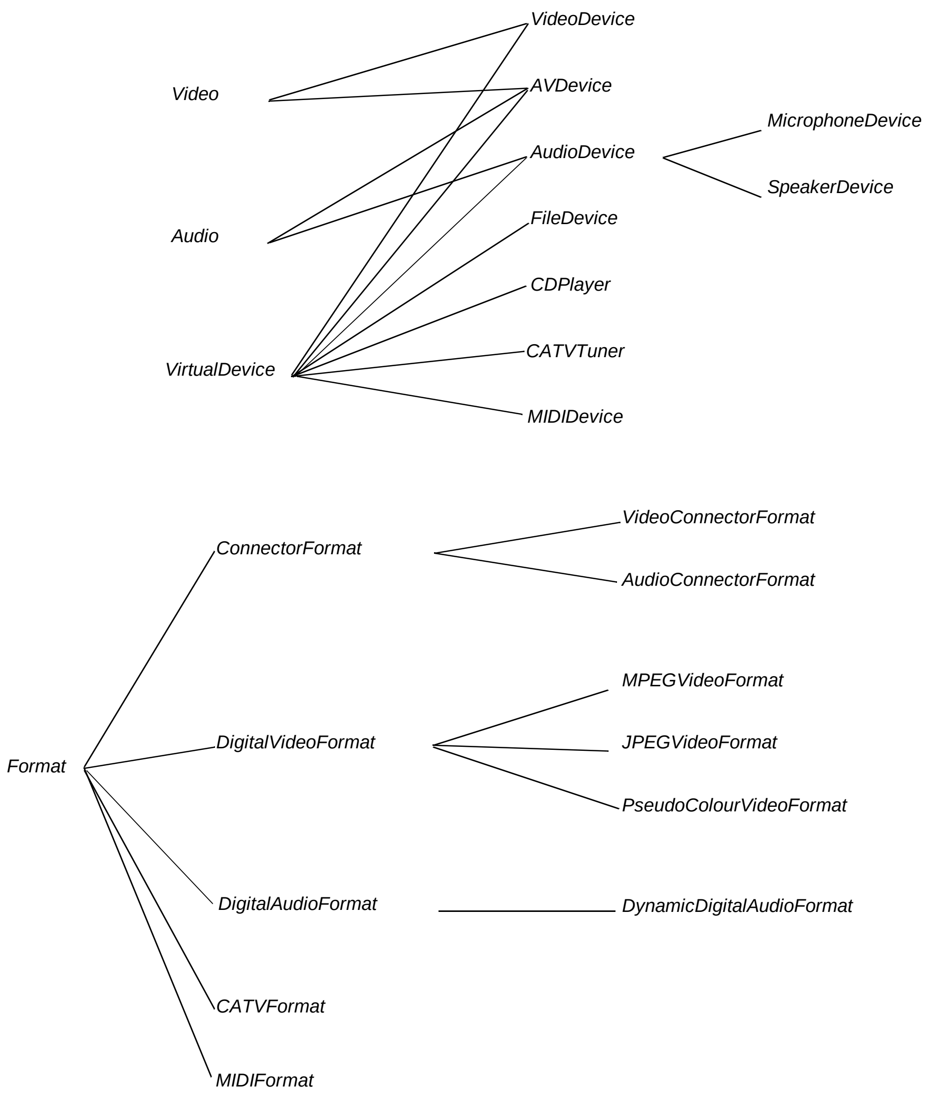


Figure 15 — Subtyping diagram for specific devices

later. For the time being, look at Figure 15 and identify the following features of the diagram to see the broad organization of the Multimedia Systems Services interfaces:

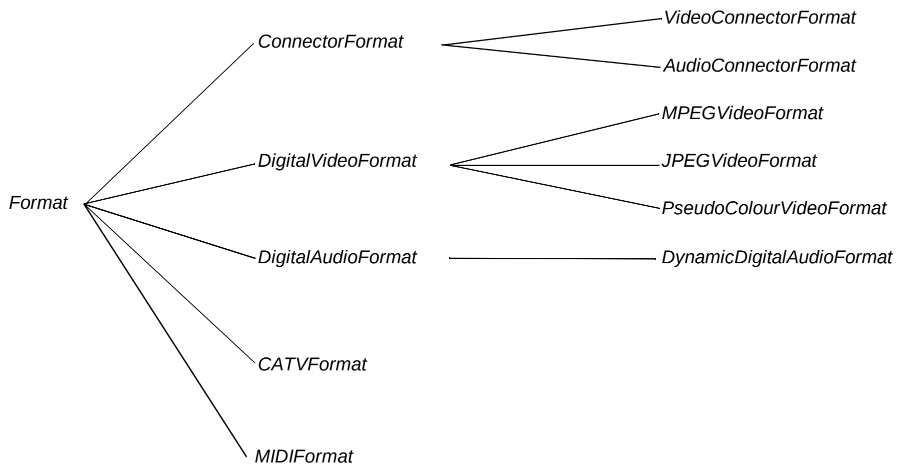


Figure 16 — *Format* subtypes

a) In the region of the diagram descending from *VirtualResource*, the descendants of *VirtualDevice* are often multiple subtypes of other types (*Audio*, *Video*, etc.) which specify operations required by real devices.

b) The diagram shows increasingly media and device specific types from left to right. These types are meant to illustrate sample types that can be supported by the Multimedia Systems Services and are not meant to be the final word in media-specific interface abstraction.

c) Note the varieties of *Format* and *Stream* interfaces. These are discussed in subsequent clauses.

C.1 Format objects

This Annex defines a hierarchy of *Format* types that can be used to describe a variety of media formats. Each *Format* type understands the details of a particular kind of media format and, through the property mechanism, has methods to set and get details about its encoding.

Figure 16 gives the hierarchy of the various format objects defined in this Annex. The *Format* objects proper acts as a common supertypes for all format objects; it defines only two general properties: byte order for the bitstream, and name of the format. All the specific features are left to the various subtypes.

The following sub–clauses give only a short overview of the various format objects defined in this Annex.

C.1.1 Video formats

This Annex defines several video *Format* types. These types are used to encapsulate the video formats which can flow between video virtual devices.

C.1.1.1 Digital video format

The *DigitalVideoFormat* type is the simplest video format. The type defines a property to describe the raster size, common colour space, and frame rate. There are properties for the video components to characterize the ordering and sub–sampling of the components.

The *PseudoColourVideoFormat* type provides information to specify the colour table for video devices with an indexed colour space. The colour table translates the index colour space into a true colour space.

The *JPEGVideoFormat* type defines the segment keys which this compression technique requires. These segment keys represent in–band control messages. For example, the key *JPEGVideoFormat::DqtK* defines the quantization table. This video format illustrates how the format object can encapsulate dynamic formats which encode commands into the data stream.

The *MPEGVideoFormat* interface defines properties for this compression technique. The list of keys is extensive. The allocation of these details to the format type, however, consolidates the conventions to one interface.

C.1.1.2 External video format

This Annex defines some additional external video format properties. These format keys anticipate the integration of device control into the framework. Assume someone provides a virtual device object which provides the type to control a VCR device. The virtual device would describe the video stream which it can export in terms of a video format object. Assume the framework also provides a video capture device. Since both virtual devices describe video formats, the client (or the connection object) can verify the formats are compatible. For example, the key "*EncodingK*" for the *VideoConnectorFormat* object may have the value of "*NTSC*", "*PAL*", "*SECAM*", "*RGB*", "*YUV*", "*AUTO*" to describe the various available formats for digital and analogue video.

C.1.2 Audio formats

This Annex offers three audio specific format types. These types are used to abstract the format of audio data that can flow across connections and be produced or consumed by audio-capable virtual devices.

C.1.2.1 Digital Audio Format

The *DigitalAudioFormat* type is intended to abstract the set of audio formats. Its characteristics are those that are necessary to fully specify a digital audio bitstream, and they include such things as encoding, sample rate, and bits per sample.

The special properties of the *DigitalAudioFormat* are such that they cannot be changed once either of the following two conditions occur:

a) The port with which it is associated is connected to another virtual device (see clause 9) via the *connect* operation; or

b) The *VirtualDevice*, *VirtualConnection*, or *Group* with which it is associated has its resources acquired via the *acquireResource* operation.

The reason for the first condition is that the data format abstracted by *DigitalAudioFormat* does not define any in–band signalling of changes in its characteristics, such as sample rate. The *VirtualConnection* establishes an agreement between the source and destination format objects; changing the value of a characteristic of one of the format objects on the fly violates this agreement and could cause data flowing through the connection to be misinterpreted.

The reason for the second condition is that the resources required to process or transport the audio data are dependent on the settings of the *DigitalAudioFormat* characteristics. Once the *VirtualResource.acquireResource* operation is invoked, a quality of service contract is established; changing the constraint on the properties of a format object may cause this contract to be violated.

C.1.2.1.1 Dynamic Digital Audio Format

The *DynamicDigitalAudioFormat* type is a subtype of the *DigitalAudioFormat* type. It defines no new methods or properties, but it is intended to abstract audio data formats that provide the in–band signalling necessary to change values of its characteristics, such as sample rate, on the fly.

Because of the in–band signalling capability, the conditions that prevented the characteristic settings of a *DigitalAudioFormat* object from being changed do not prevent the characteristic settings of a *DynamicDigitalAudioFormat* object from being changed. However, the client should be careful to constrain the range of characteristic settings to those it expects to use in order to improve the chances of a successful connection and to prevent over–allocation of resources.

C.1.2.2 Audio Connector Format

The *AudioConnector* interface abstracts audio formats that are made external to a computer system, such as line, headphones, and microphone level analogue signals. The properties of this format are used to determine the compatibility of data types in the same way as for internal digital audio data types. For example, it is possible to determine that a port that uses a line level cannot be connected to a port that uses a headphones level.

The *AudioConnectorFormat* is an example of a *Format* type that is associated with *VirtualDevice* ports that model physical connections to a device. Exposing the physical connectors of a device as ports and describing the data that can flow across them as *Formats* permits external devices to be modelled in exactly the same way as internal devices.

C.1.3 CATV format

CATV is the acronym for Cable TV. The *CATVFormat* object gives a standardized way to describe access to CATV network information.

C.1.4 MIDI format

MIDI is the acronym for Musical Instrument Digital Interface, used to interface synthesizers, sequencers, rhythm machines, etc. All MIDI communication is achieved through multi–byte messages which are defined in separate specifications. The *MIDIFormat* object gives a standardized way to interface such devices.

C.2 Digital stream controls

All devices, described in this Annex, use digital stream control. This means that the generic *SyncStream* and *SyncStreamControl* objects, as defined in clause 8, are actualized so that their progression space is defined to be Z_∞, and all virtual device objects in this Annex used these objects instead of the general ones.

C.3 Video and audio processing

C.3.1 Video processing

The basic video specific processing interface is defined by the *Video* type. This is an abstract base type that is “mixed in” with the *VirtualDevice* type through multiple subtyping to create the *VideoDevice* interface.

The *Video* type was made a distinct type itself, rather than defining its methods directly in the *VideoDevice* type defined later, in order to allow adding video processing operations to other types of devices (such as an audio/video device) without encountering ambiguous multiple subtyping. This approach ensures that there is exactly one path from a *VirtualDevice* subtype to its base types.

The interface of the type allows the client to manipulate standard video controls. Since devices often provide additional controls, the interface casts the controls as a property list. With this syntax it is straightforward to define additional *KeyValue* pairs, and still inherit the operations found in the video type. Note that since the operations includes the stream position, the client can enqueue methods which modify the controls as a function of the steam time.

The interface of the *Video* type provides an operation to set the *Area Of Interest* within the source raster. The structure to define an area of interest describes the origin *(x,y)* and size *(width,height)* within the source raster which the object is to process. Note that the width and height can assume negative values. The four combinations of positive and negative values allow the client to reflect the image across either or both of the vertical and horizontal axes.

The operation which specifies the area of interest (*setImageAOI*) also takes a stream position (i.e., stream relative time) argument to allow for changes in the area of interest as the stream flows. The current value of the area of interest can also be inquired. Also, since the region within the source raster can be larger than the target raster, there is a property to specify whether the object is to clip or scale.

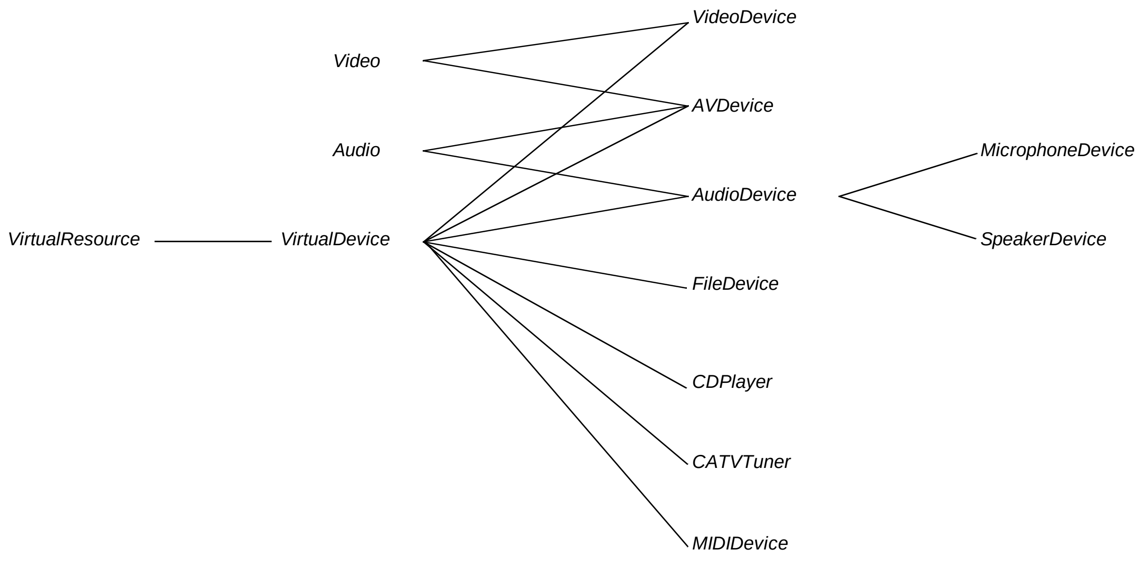


Figure 17 — *VirtualDevice* subtyping hierarchy

C.3.2 Audio processing

The basic audio specific processing interface is defined by the *Audio* type. This is an abstract base type that is “mixed in” with the *VirtualDevice* type through multiple subtyping to create the *AudioDevice* interface.

The *Audio* type was made a distinct type itself, rather than defining its methods directly in the *AudioDevice* type defined later, in order to allow adding audio processing operations to other types of devices (such as an audio/video device) without encountering ambiguous multiple subtyping. This approach ensures that there is exactly one path from a *VirtualDevice* subtype to its base types.

The interface of *Audio* is very simple: it only defines a properties for both the gain and the attenuation of the signal from the input(s) to the output(s) of an audio device.

C.4 Specific devices

In this subclause, several useful subtypes of *VirtualDevice* will be described:

a) video,

b) audio,

c) file, and

d) external device.

These types will demonstrate how abstractions of real world devices can be realized within the Multimedia Systems Services model. The types used here are examples only. Additional devices may be defined by registration.

C.4.1 Defining a device

When adding a new device type, it is necessary to evaluate the device's capabilities with respect to the following criteria:

a) What processing does the device perform on its data?

b) What types of data can be processed? What characteristics define them?

c) What stream position and control vocabulary is needed?

The answer to the first question determines the new operations that the new *VirtualDevice* subtype should have. The answer to the second question determines what new *Format* and possibly *Protocol* type(s) are necessary to configure the ports. The answer to the third question determines if any new *StreamControl* subtype or position keys may be needed.

In addition, one should consider what the configuration of the device's ports should be. Typically, each point to which a connection can be made (either virtual or physical) should be exposed as a virtual device port. For example, an audio digitizer device that can input data from a line input and produce a stream of digital data would have one input and one output port; in contrast, a speaker device that can input data from a digital stream and produce sound would have one input port and no output ports.

C.4.2 Video

Most of the general characteristics of video processing are abstracted in the separate *Video* object type (see Annex C.3 of this Part), as well as in the various video related *Format* types (see Annex C.1 of this Part). No new stream control vocabulary is necessary. This Annex of PREMO defines only one *VideoDevice* type; the definition of specific subtypes for various operating environments would go beyond the scope of this Annex and should refer to the special features of these environments.

NOTE — Informal ISO documents or appendices to this standard may however be published to give a general highlight of how this could be done for specific environments.

C.4.3 Audio

The addition of audio devices to this Annex involves the definition of audio specific processing capabilities and media formats. No new stream control vocabulary is necessary.

C.4.3.1 Audio processing

The basic audio specific processing interface is defined by the *Audio* interface (see Annex C.3.2 of this Part). This type is an abstract base type that is “mixed in” with the *VirtualDevice* type to create the *AudioDevice* type. In addition, certain characteristics specific to audio processing are defined, such as the encoding and sample rate associated with the inputs and outputs; this defines a vocabulary sufficient for the client to specify simple constraints about the type of *AudioDevice*, such as “an Audio Device that can process 44.1 KHz at its input and produce 8 KHz at its output”.

The *AudioDevice* interface can be used to represent any device that processes audio and supports the gain and attenuation properties. If a device is incapable of affecting the gain, it can either be represented as an *AudioDevice* with its gain value fixed at unity, or simply as a *VirtualDevice* that admits to audio *Format* objects at its ports (see below).

C.4.3.1.1 Speaker device

The *SpeakerDevice* is a subtype of the *AudioDevice* type. It is used to abstract a built–in speaker device. This device presents one input port and has no output ports (it just pushes air). It defines no new operations.

The *SpeakerDevice* type is defined to make an easy distinction for the client between this type of device and, say, an audio display device that drives a line output. Both can be equally well represented by the *AudioDevice* interface. The *SpeakerDevice* type provides a simple way for the client to get a speaker, without having to set the appropriate constraints on a generic *AudioDevice*.

C.4.3.1.2 Microphone device

The *MicrophoneDevice* is a subtype of *AudioDevice*. It is used to abstract a built–in microphone device. It presents one output port and no input ports. Like *SpeakerDevice*, it is defined as a simple way for a client to obtain a microphone, without having to set the appropriate constraints on a generic *AudioDevice*.

C.4.4 Files

A pervasive characteristic of multimedia applications is the use of files to store and play back media. Defining a separate device to describe file processing in a multimedia setting gives a high level of uniformity to multimedia applications.

Common to the processing of most media file types is a core set of operations required by clients for the playback and storage of media data. These operations include:

a) setting the mode for file access, e.g., read, write;
b) identifying the file format type;
c) inquiry on file and data characteristics;
d) setting the position within the file for data access;
e) setting ranges within the file to constrain data access;
f) repetitive data access; and
g) controlling the speed of playback.

Files introduce no new media formats, but rather use the same *Format* objects as already defined for the media-specific device (i.e., *DigitalAudioFormat*, *DigitalVideoFormat*, etc.). The operations unique to particular file formats are supplied by subtyping the *VirtualDevice* type.

The specific characteristics of a file virtual device are determined by the file format type which it represents and its mode of operation, e.g., reading or writing. Several approaches are possible for managing the process of discovering the file's type and launching the appropriate file reader or writer device. One fairly simple strategy is to assume that the client has knowledge, by some means, of the type of file it wishes to read or write. The client creates a file virtual device by first locating a factory that can provide an object of the desired *FileDevice* subtype, then creates an object on the factory with a constraint list that describes the client's requirements and knowledge of the file. The constraint list minimally includes a file name, open mode, and file type, in addition to any constraints required by supertypes. It may optionally include a location, a temporary file name, and an “automatic save on close” indicator.

A variation on this approach could utilize a special factory to perform the task of creating *FileDevice* objects, rather than the client. Still other approaches could easily be described for handling complex file types, such as container files, but this is beyond the scope of the this standard.

Whichever approach is used, as part of the creation process, a subtype of *FileDevice*, identified by the file type, is instantiated and the file device configures itself with the appropriate number and type of ports and associated *Format* and *FileStream* objects. Ports are designated for input or output according to the open mode constraint. Formats are defined according to information found in the file, e.g., a header that describes the media characteristics of the data. The client may inquire about the nature of the ports configured by using the *VirtualDevice.getPortConfig* method. This returns a list of ports which includes associated format classes and input/output designations for each port. The client can use this information to decide which port to designate when connecting to another virtual device.

C.4.5 CD player

The *CDPlayer* is a subtype of the *Device* object, used to abstract the access to compact disk players. It does not use any particular format objects.

C.4.6 CATV tuner

The *CATVTuner* is a subtype of the *Device* object, used to abstract the access to cable TV information. Its ports are controlled by the *CATVFormat* objects (see Annex C.1.3 of this Part).

C.4.7 MIDI device

The *MIDIDevice* is a subtype of the *Device* objects, whose input and output ports are controlled by the *MIDIFormat* objects (see Annex C.1.4 of this Part). It is used to abstract access to various MIDI hardware, like synthesizers, rhythm machines, etc. It may have several input and several output ports.

C.4.8 External resources

An important aspect of multimedia in any computer environment, distributed or otherwise, is control of external devices. External devices such as cameras, microphones, speakers, etc. provide the “eyes” and “ears” and “mouth” of a computer system, transducing real world analog signals to digital signals for communication or storage, and transducing digital signals to analog signals for interaction with people. External devices such as VCRs and laser disc players provide source material and inexpensive mass storage for analog information; TV or radio tuners provide access to additional analog sources. The list of useful external devices is long, and the Multimedia Systems Services must allow for external device attachment and control.

The abstraction for external devices is the *VirtualDevice*. As with all virtual devices, the inputs and outputs for external devices are exposed as ports, which have associated *Format* objects. Since ports on external devices are used to represent external connectors, the generic *Format* type is subtyped from *ConnectorFormat*. The *ConnectorFormat* type (see Annex C.5.2.1 of this Part) defines those characteristics specific to the type of data that passes through the port. It could be used to represent a physical connector on an external device or an internal adapter card, such as an audio capture device.

By extending the same abstraction used for “internal” devices to “external” devices, the Multimedia Systems Services allows the same sort of actions on external devices as on internal devices. For instance, external connectors can be connected (although some connections may be hard wired), external formats can be constrained, queried, etc., in the same manner as for internal devices.

As an example, consider the *CATVTuner* in clause C.4.6. This device takes an RF analog video input and generates two channels of analog audio (stereo) output plus one channel of analog video output. Thus, the *CATVTuner* admits to one input port and three output ports all of which have *ConnectorFormats*. The input format would actually be a *CATVFormat*, representing the RF analog video. The audio output channels would each be represented by an *AudioConnectorFormat*. The video output channel would be represented by a *VideoConnectorFormat*.

Next consider a video capture adapter. Its output port would be represented by a *DigitalVideoFormat*. Its input connector would be represented by a *VideoConnectorFormat*. With this design, the *CATVTuner* video output port could be successfully connected to a video capture device input port to allow capture of the video output of the tuner. It is possible to connect the *CATVTuner* audio connectors to one or more audio capture adapters making similar assumptions.

It should be noted that there are a variety of ways to realize virtual connections between external devices. Some installations may have physical device inputs and outputs routed to a switching device, such as an analog crossbar switch. When a *VirtualConnection* object is created, the resource manager controlling the crossbar switch could be instructed to connect the appropriate ports. Another realization might simply be to display a message on a user's screen and ask that the appropriate connection be made. Finally, “persistent” connections could be stored in the Registration & Retrieval Service database, permitting a virtual connection factory to determine whether a reference to a requested virtual connection could be made.

C.5 Functional Specification

This clause gives a more formal specification of the objects defined in this Annex. Note that the specifications are here as example only, hence some of the keys and properties are not fully specified.

C.5.1 Area of interest for video objects

ImageAOI

SimplePREMOObject

x, y, width, height: $\mathbb{R}$

ImageAOI

C.5.2 Format objects

C.5.2.1 *ConnectorFormat* objects

$ConnectorFormat_{abtract}$

Format

ConnectorFormat

C.5.2.2 *VideoConnectorFormat* object

VideoConnectorFormat

ConnectorFormat

VideoConnectorFormat

Properties defined:

Key	**Type of Value**	**R.O or R/W**	**Description**
EncodingK	String	R/W	Shorthand name which identifies the video encoding or syntax.
VideoSyncStateK	*String*	R/W	Indicates whether video is synchronized.
VideoSyncSourceK	*String*	R/W	Indicate the source of the video synchronization signal.

Capabilities defined:

Key	**Type of Value**	**Values**
EncodingCK	array_6 String	<*"NTSC", "PAL", "SECAM", "RGB", "YUV", "AUTO"*>
VideoSyncStateCK	array_3 String	<*"GoodSync", "InvalidSync", "NoSync"*>
VideoSyncSourceCK	array_3 *String*	<*"InternalSync", "ExternalSync", "AutoSync"*>
MutablePropertyListCK	seq *Key*	<*"EncodingK", VideoSyncStateK" "VideoSyncSourceK"*>

C.5.2.3 ***AudioConnectorFormat*** objects

AudioConnectorFormat

ConnectorFormat

AudioConnectorFormat

Properties defined:

Key	**Type of Value**	**R.O or R/W**	**Description**
EncodingK	String	R/W	

Capabilities defined:

Key	**Type of Value**	**Values**
EncodingCK	array$_4$ String	<*"Headphone"*, *"Line"*, *"Microphone"*, *"Speaker"*>

C.5.2.4 *DigitalVideoFormat* object

DigitalVideoFormat$_{abtract}$

Format

DigitalVideoFormat

Properties defined:

Key	Type of Value	R.O or R/W	Description
WidthK	$\mathbf{Z}$	R.O.	
HeightK	$\mathbf{Z}$	R.O.	
AspectRatioK	$\mathbf{R}$	R.O.	
ColourSpaceK	*String*	R.O.	
PeriodK	$\mathbf{R}$	R.O.	
SampleRateK	$\mathbf{R}$	R.O.	
ComponentCountK	$\mathbf{Z}$	R.O.	
C1TypeK	*String*	R.O.	
C1DepthK	*Octet*	R.O.	
C1SubsamplingXK	*Octet*	R.O.	
C1SubsamplingYK	*Octet*	R.O.	
C2TypeK	*String*	R.O.	
C2DepthK	*Octet*	R.O.	
C2SubsamplingXK	*Octet*	R.O.	
C2SubsamplingYK	*Octet*	R.O.	
C3TypeK	*String*	R.O.	
C3DepthK	*Octet*	R.O.	
C3SubsamplingXK	*Octet*	R.O.	
C3SubsamplingYK	*Octet*	R.O.	
C4TypeK	*String*	R.O.	
C4DepthK	*Octet*	R.O.	
C4SubsamplingXK	*Octet*	R.O.	
C4SubsamplingYK	*Octet*	R.O.	

Capabilities defined:

Key	**Type of Value**	**Values**
WidthCK	$\mathbf{Z} \times \mathbf{Z}$	<minimum, maximum>
HeightCK	$\mathbf{Z} \times \mathbf{Z}$	<minimum, maximum>
PeriodCK	$\mathbf{R} \times \mathbf{R}$	<minimum, maximum>
AspectRatioCK	$\mathbf{R} \times \mathbf{R}$	<minimum, maximum>
ColourSpaceCK	array_6 *String*	*<"RGB_LINEAR", "RGB_709", "YCC_LINEAR", "YCC_709", "Y_LINEAR", "Y_709">*
MutablePropertyListCK	seq *Key*	*<"WidthK", "HeightK", "AspectRatioK", "PeriodK", "SampleRateK", "ComponentCountK", "C1TypeK", "C1DepthK", "C1SubsamplingXK", "C1SubsamplingYK", "C2TypeK", "C2DepthK", "C2SubsamplingXK", "C2SubsamplingYK" "C3TypeK", "C3DepthK", "C3SubsamplingXK", "C3SubsamplingYK" "C3TypeK", "C3DepthK", "C3SubsamplingXK", "C3SubsamplingYK">*

C.5.2.5 ***MPEGVideoFormat*** **object**

MPEGVideoFormat

DigitalVideoFormat

MPEGVideoFormat

Properties defined:

Key	**Type of Value**	**R.O or R/W**	**Description**
IntraQMatrixK	array_8(array$_8$ Octet)	R/W	
InterQMatrixK	array_8(array$_8$ Octet)	R/W	
TimeCodeK	array$_5$ *Octet*	R/W	The values are (in this order): a *drop-Flag*, hours, minutes, seconds, pictures. A *dropFlag* value of zero is interpreted as TRUE and any other value is inter-preted as FALSE.

Capabilities defined:

Key	**Type of Value**	**Values**
MutablePropertyListCK	seq *Key*	<*"IntraQMatrixK", "InterQMatrixK" "TimeCodeK"*>

C.5.2.6 ***JPEGVideoFormat*** **object**

JPEGVideoFormat

DigitalVideoFormat

JPEGVideoFormat

Properties defined:

None.

Capabilities defined:

None.

C.5.2.7 ***PseudoColourVideo*** **object**

PseudoColourVideoFormat

DigitalVideoFormat

PseudoColourVideoFormat

Properties defined:

Key	Type of Value	R.O or R/W	Description
NumColoursK	$\mathbb{Z}$	R.O.	
ColourTableK	seq ($\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$)	R/W	The values are (in consecutive order): index, red, green, blue, and alpha values.

Capabilities defined:

Key	Type of Value	Values
DynamicPropertyListCK	seq *Key*	<“ *ColourTableK*” >

C.5.2.8 *DigitalAudioFormat* object

See also the restrictions on changing property values, described in Annex C.4.3.1 of this Part.

DigitalAudioFormat

Format

DigitalAudioFormat

Properties defined:

Key	Type of Value	R.O or R/W	Description
EncodingK	*String*	R/W	Shorthand name which identifies the audio encoding or syntax.
SampleRateK	*String*	R/W	
NumChannelsK	$\mathbb{Z}$	R.O.	
SampleSizeK	$\mathbb{Z}$	R/W	Size of a sample in bits. A zero value indicates variable.
SignedK	*Boolean*	R.O.	Refers to the interpretation of the sample values. The type of the native property value is seq *Boolean*.

Capabilities defined:

Key	Type of Value	Values
EncodingCK	array$_4$ String	*<"ALAW", "ULAW", "LINEAR", "ADPCM">*
SampleRateCK	array$_5$ String	*<"8KHz", "11.03KHz", "22.05KHz", "40.0KHz", "44.1KHz">*
SampleSizeCK	seq($\mathbf{Z} \times \mathbf{Z}$)	seq*(minimum* $\times$ *maximum)* (One pair per channel)
MutablePropertyListCK	seq *Key*	*<"EncodingK", "SampleRateK", "SampleSizeK", "SignedK">*

C.5.2.9 *DynamicDigitalAudioFormat* object

See also the restrictions on changing property values, described in Annex C.1.2.1.1 of this Part. This type does not add any operation nor does it add new properties or capabilities; the type is defined separately to stress the semantic differences.

DynamicDigitalAudioFormat

DigitalAudioFormat

DynamicDigitalAudioFormat

C.5.2.10 *CATVFormat* object

This type does not add any operation nor does it add new properties or capabilities; the type is defined separately to stress the semantic differences. The format is relevant with relation to the *CATVTuner* device.

CATVFormat

Format

CATVFormat

C.5.2.11 *MIDIFormat* object

MIDIFormat

Format

MIDIFormat

C.5.3 Digital Stream Control

C.5.3.1 *DigitalStreamControl* objects

DigitalStreamControl

StreamControl$[Z_\infty]$

DigitalStreamControl

C.5.3.2 *DigitalStreamControl* objects

DigitalSyncStreamControl

SyncStreamControl$[Z_\infty]$

DigitalSyncStreamControl

C.5.4 Video and audio processing

C.5.4.1 *Video* object

Video$_{abstract}$

PropertyInquiry

setImageAOI

position$_{in}$: *Time*
aoi$_{in}$: *RefImageAOI*
exceptions: {*InvalidPosition*}

The operation allows the client to specify the area of interest within the source image. The default value is the entire source raster. *aoi*$_{in}$ describes the origin *(x, y)* and size *(width, height)* within the source image which is to be processed. The width and height can assume negative values. The four possible sign combinations allow the client to reflect the image across either or both of the vertical (negative height value) and horizontal (negative weight value) axes, or to specify no reflection (both weight and height positive.) *position*$_{in}$ is a stream position. Using it, the client can specify that the AOI is to be effective at and following the specified steam time.

Exceptions raised:

InvalidPosition	*position*$_{in}$ is not valid.

getImageAOI

position$_{in}$: *Time*
aoi$_{out}$: *RefImageAOI*[Shallow Copy]
exceptions: {*InvalidPosition*}

The operation returns the area of interest at the specified stream time.

Exceptions raised:

InvalidPosition	*position*$_{in}$ is not valid.

Video

Properties defined:

Key	Type of Value	R.O or R/W	Description
BrightnessK	$\mathbb{Z}$	R/W	
ContrastK	$\mathbb{Z}$	R/W	
SaturationK	$\mathbb{Z}$	R/W	
SharpnessK	$\mathbb{Z}$	R/W	
ImageAOIModeK	String	R/W	

Capabilities defined:

Key	**Type of Value**	**Values**
BrightnessCK	$\mathbb{Z} \times \mathbb{Z}$	<minimum value, maximum value>
ContrastCK	$\mathbb{Z} \times \mathbb{Z}$	<minimum value, maximum value>
SaturationCK	$\mathbb{Z} \times \mathbb{Z}$	<minimum value, maximum value>
SharpnessCK	$\mathbb{Z} \times \mathbb{Z}$	<minimum value, maximum value>
ImageAOIModeCK	array$_2$ String	*<"Clip", "Scale">*
DynamicPropertyListCK	seq *Key*	*<"BrightnessK", "ContrastK", "SaturationK", "SharpnessK", "ImageAOIModeK">*

C.5.4.2 *Audio* object

Audio$_{abstract}$

PropertyInquiry

Audio

Properties defined:

Key	**Type of Value**	**R.O or R/W**	**Description**
GainK	$\mathbb{R}$	R/W	Sound Volume

Capabilities defined:

Key	**Type of Value**	**Values**
DynamicPropertyListCK	seq *Key*	*<"GainK">*

C.5.5 Specific devices

C.5.5.1 *VideoDevice* object

VideoDevice

VirtualDevice, *Video*

VideoDevice

Properties defined:

Key	Type of Value	R.O or R/W	Description
InputFrameRateK	$\mathbf{R}$	R/W	
OutputFrameRateK	$\mathbf{R}$	R/W	

Capabilities defined:

Key	Type of Value	Values
InputFrameRateCK	$\mathbf{R} \times \mathbf{R}$	<minimum, maximum>
OutputFrameRateCK	$\mathbf{R} \times \mathbf{R}$	*<minimum, maximum>*
InputPortCK	$\mathbf{Z}$	*1*
OutputPortCK	$\mathbf{Z}$	*1*
InputFormatTypesCK	seq *ObjectType*	*<DigitalVideoFormat>*
OutpuFormatTypesCK	seq *ObjectType*	*<DigitalVideoFormat>*

C.5.5.2 *AudioDevice* object

AudioDevice

VirtualDevice, *Audio*

AudioDevice

Capabilities defined:

Key	Type of Value	Values
InputPortCK	$\mathbf{Z}$	*1*
OutputPortCK	$\mathbf{Z}$	*1*
InputFormatTypesCK	seq *ObjectType*	*<DigitalAudioFormat>*
OutpuFormatTypesCK	seq *ObjectType*	*<DigitalAudioFormat>*

C.5.5.3 *AVDevice* object

This type represents A/V multiplexors, demultiplexors, A/V capture and display devices.

AVDevice

VirtualDevice, Video, Audio

AVDevice

Capabilities defined:

Key	Type of Value	Values
InputPortCK	***Z***	*2*
OutputPortCK	***Z***	*2*
InputFormatTypesCK	seq *ObjectType*	*<DigitalVideoFormat, DigitalAudioFormat>*
OutpuFormatTypesCK	seq *ObjectType*	*<DigitalVideoFormat, DigitalAudioFormat>*

C.5.5.4 *MicrophoneDevice* object

The *MicrophoneDevice* object is a subtype of *AudioDevice*. It is defined just to make it easy for a client to get a microphone, rather than specify the necessary constraints on *AudioDevice*.

MicrophoneDevice

AudioDevice

MicrophoneDevice

Capabilities defined:

Key	Type of Value	Values
InputPortCK	***Z***	*0*
OutputPortCK	***Z***	*1*
InputFormatTypesCK	seq *ObjectType*	*<>*
OutpuFormatTypesCK	seq *ObjectType*	*<DigitalAudioFormat>*
GlobalFormatTypesCK	seq *ObjectType*	*<QoSDescriptor, AudioConnectorFormat>*

C.5.5.5 *SpeakerDevice* object

The *SpeakerDevice* object is a subtype of *AudioDevice*. It is defined just to make it easy for a client to get a speaker, rather than specify the necessary constraints on *AudioDevice*.

SpeakerDevice

AudioDevice

SpeakerDevice

Capabilities defined:

Key	Type of Value	Values
InputPortCK	***Z***	*1*
OutputPortCK	***Z***	*0*
InputFormatTypesCK	seq *ObjectType*	*<DigitalAudioFormat>*
OutpuFormatTypesCK	seq *ObjectType*	*<>*
GlobalFormatTypesCK	seq *ObjectType*	*<QoSDescriptor, AudioConnectorFormat>*

C.5.5.6 *FileDevice* object

FileDevice

VirtualDevice

save

exceptions: {*InvalidAccess*}

Save the file content.

Exceptions raised:

InvalidAccess	Invalid access rights to the file.

FileDevice

Properties defined:

Key	**Type of Value**	**R.O or R/W**	**Description**
NameK	String	R/W	
OpenModeK	String	R/W	
TypeK	String	R/W	
AutoSaveK	String	R/W	

Capabilities defined:

Key	**Type of Value**	**Values**[a]
OpenModeCK	seq *String*	<*"Read Mode", "Insert Mode", "OverWrite Mode", "Append Mode", "Create Mode"*>
DynamicPropertyListCK	seq *Key*	<*"AutoSaveK"*>
InputPortCK	$\mathbb{Z}$	*1*
OutputPortCK	$\mathbb{Z}$	*1*
InputFormatTypesCK	seq *ObjectType*	<*Format*>
OutpuFormatTypesCK	seq *ObjectType*	<*Format*>

a. an instance of FileDevice may have either one output port or one input port, but not both (this is reflected in the native property values for the properties InputPortC and OutputPortC)

C.5.5.7 *CDPlayer* object

CDPlayer

VirtualDevice

CDPlayer

Properties defined:

Key	**Type of Value**	**R.O or R/W**	**Description**
TrackK	$\mathbb{Z}$	R.O	Current track
TrackDurationK	seq *Time*	R.O.	Duration of each track
TrackCountK	$\mathbb{Z}$	R.O.	Number of tracks
PlayListK	seq $\mathbb{Z}$	R/W	List of tracks.

Capabilities defined:

Key	**Type of Value**	**Values**
DynamicPropertyListCK	seq *Key*	*<"PlayListK">*
InputPortCK	$\mathbb{Z}$	*0*
OutputPortCK	$\mathbb{Z}$	*2*
InputFormatTypesCK	seq *ObjectType*	*<>*
OutpuFormatTypesCK	seq *ObjectType*	*<DigitalVideoFormat, DigitalAudio-Format>*

C.5.5.8 *CATVTuner* object

This device models a CATV tuner that takes an RF analog video at its input and generates two channels of audio plus one channel of video as output. Thus, it has one input port and three output ports.

CATVPlayer

VirtualDevice

CATVPlayer

Properties defined:

Key	Type of Value	R.O or R/W	Description
NumberOfChannelsK	***Z***	R.O	
CurrentChannelK	***Z***	R/W	

Capabilities defined:

Key	Type of Value	Values
DynamicPropertyListCK	seq *Key*	*<"CurrentChannelK">*
InputPortCK	***Z***	*1*
OutputPortCK	***Z***	*3*
InputFormatTypesCK	seq *ObjectType*	*<CATVFormat>*
OutpuFormatTypesCK	seq *ObjectType*	*<DigitalVideoFormat, DigitalAudio-Format>*
GlobalFormatTypesCK	seq *ObjectType*	*<QoSDescriptor, VideoConnectorFor-mat>*

C.5.5.9 *MIDIDevice* object

This device models a MIDI device, to access synthesizers, rythm machines, etc. It may have several input and output ports.

MIDIDevice

VirtualDevice

MIDIDevice

Properties defined:

Key	Type of Value	R.O or R/W	Description
NumberOfChannelsK	***Z***	R.O	“Channel” is the term used in the MIDI specification, and can be identified with various ports in the device.
Omni	Boolean	R/W	When *TRUE*, it enables to receive voice messages in all voice channels without discrimination.
Mono	Boolean	R/W	When *TRUE*, it restricts the assignment of voices to just one voice per channel. When *FALSE*, any number of voices may be allocated.

Capabilities defined:

Key	Type of Value	Values
InputPortCK	***Z***	*16*
OutputPortCK	***Z***	*16*
InputFormatTypesCK	seq *ObjectType*	*<MIDIFormat>*
OutpuFormatTypesCK	seq *ObjectType*	*<MIDIFormat>*
GlobalFormatTypesCK	seq *ObjectType*	*<QoSDescriptor, MIDIFormat>*
MutablePropertyListCK	seq *Key*	*<“Omni”>*
DynamicPropertyListCK	seq *Key*	*<“Mono”>*

Annex D
(informative)
Examples of virtual connection settings

What follows are some examples of how the virtual connections can be configured to support the types of connections defined in 9.3.

D.1 Hardware connection example

In some cases, system hardware or operating system software performs the media data transport between the virtual devices. The virtual connection determines this, but it is then the responsibility of the virtual devices to carry this out. Figure 18 shows an example; the shaded area in the figure denotes a machine boundary.

D.2 Direct connection example

The direct connection case shown in Figure 19 is typical of a stand-alone system and will also be used when possible for optimization in a distributed environment.

D.3 Local connection example

The local connection configuration is illustrated in Figure 20. Here a virtual connection adapter is instantiated and inserted by the virtual connection between two virtual devices on the same system. This might be necessary when the virtual devices ports are incompatible.

In the local virtual connection adapter case, the virtual connection adapter provides two ports. The virtual device port never knows whether it is working with another virtual device port or a virtual connection adapter.

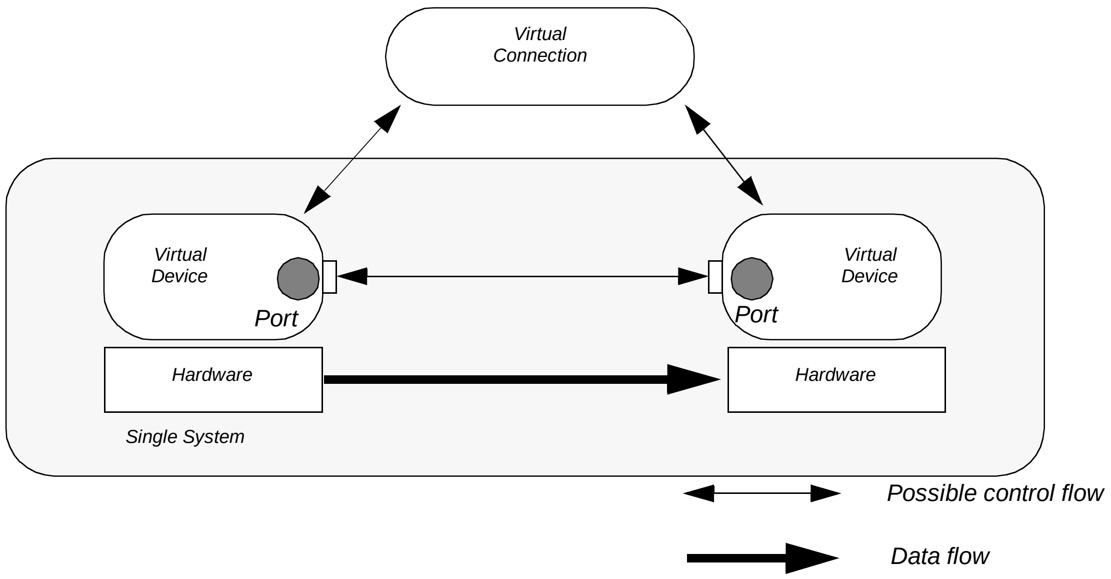


Figure 18 — Hardware connection example

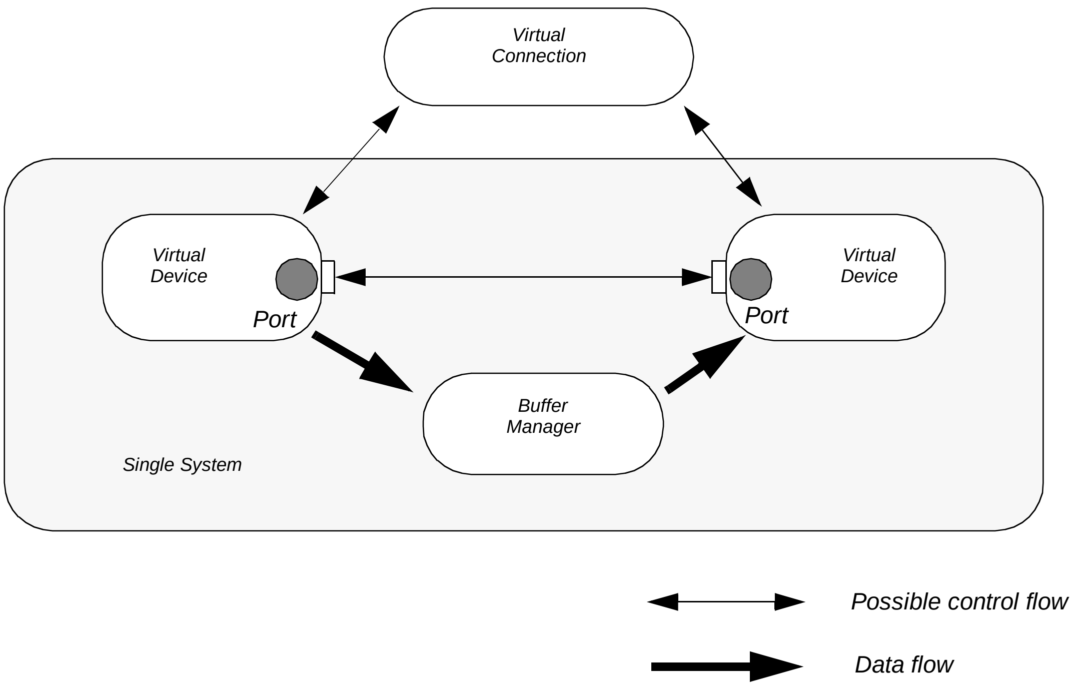


Figure 19 — Direct connection example

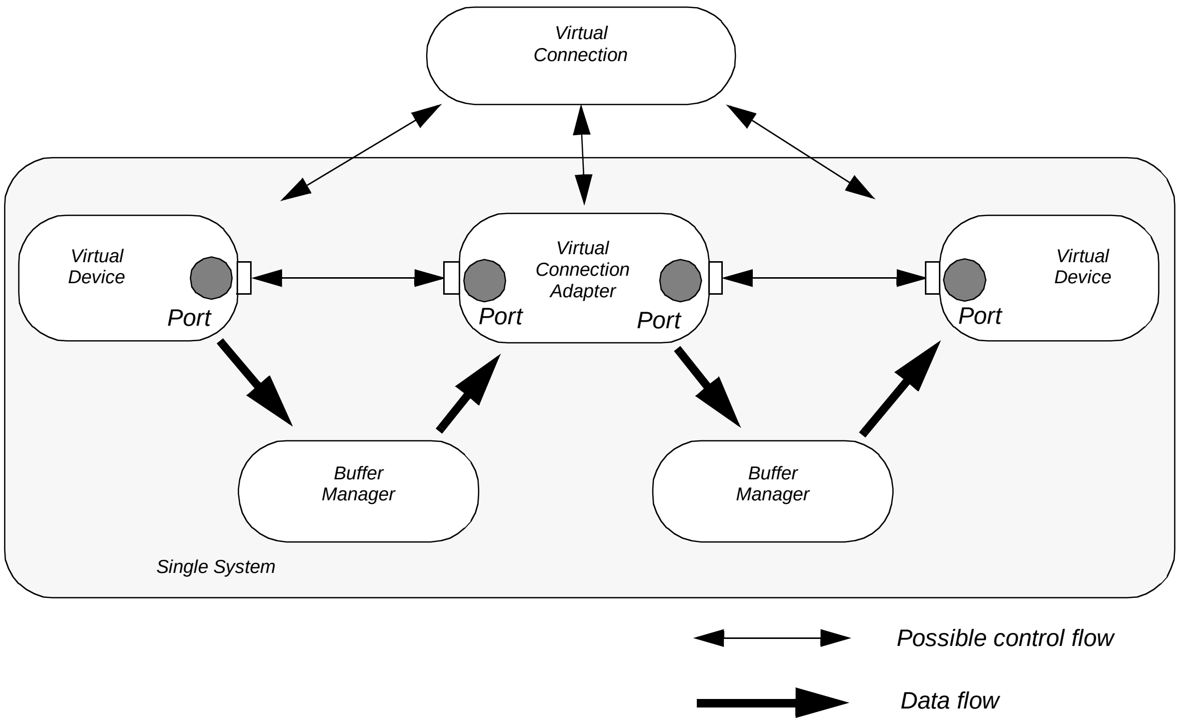


Figure 20 — Local virtual connection adapter example

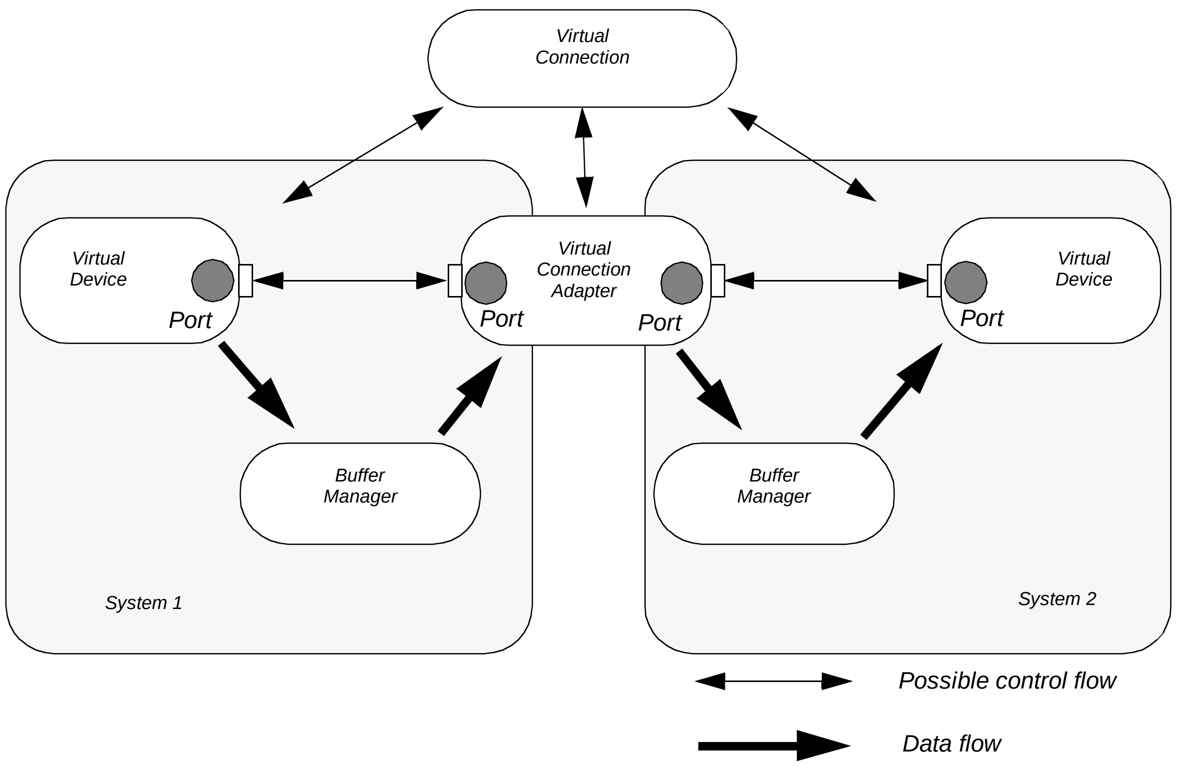


Figure 21 — Network virtual connection adapter example

D.4 Network connection example

In a distributed environment, the two virtual devices reside on separate systems and a virtual connection adapter is required. Unlike the local virtual connection adapter, which may be implemented as a single entity, the network virtual connection adapter consists of two separate entities which must communicate across the network. This communication between the two parts of the virtual connection adapter will include the media transfer as well as control information. This communication will be accomplished on top of a network protocol.