The HAVi Specification

Specification of the Home Audio/Video Interoperability (HAVi) Architecture

HAVi, Inc.

Version 1.1
May 15, 2001

Table of Contents

APPENDIX A: HAVi JAVA APIs

List of Figures

List of Tables

Detailed Table of Contents

1 General

1.1 Scope

This document provides a specification of the *Home Audio/Video Interoperability Architecture* (also called the *HAVi Architecture*). The HAVi Architecture is intended for implementation on Consumer Electronics (CE) devices and computing devices; it provides a set of services which facilitate interoperability and the development of distributed applications on home networks. HAVi is intended for, but not restricted to, CE devices supporting the IEEE Std 1394-1995 [3] (and future extensions) and IEC 61883 [4] interface standards.

Since a goal of the HAVi Architecture is to be future-proof, interoperability is more than a common command set. HAVi is a software architecture that allows new devices to be integrated into the home network and to offer their services in an open and seamless manner. The HAVi Architecture provides:

- a set of software elements along with the protocols and APIs needed to achieve interoperability
- device abstraction and device control models
- an addressing scheme and lookup service for devices and their resources
- an open execution environment supporting visual presentation and control of devices, and providing runtime support for third party applications
- communication mechanisms for extending the environment dynamically through plug-and-play capabilities
- a versioning mechanism that preserves interoperability as the architecture evolves
- management of isochronous data streams

This document describes the constructs HAVi implements to support interoperability. Specific topics covered include: the system and device models of the HAVi Architecture, the APIs and protocols used by software elements of the HAVi Architecture, and APIs for specific devices. The types of devices supported by HAVi include: tuner, VCR, clock, camera, AV disc, display, amplifier, modem, and Web proxy.

1.2 References

[1] ISO/IEC 13213:1994 Control and Status Register (CSR) Architecture for Microcomputer Buses (IEEE Std 1212-1994).

[2] IEEE P1212 Draft 1.0, Draft Standard for a Control and Status Registers (CSR) Architecture for Microcomputer Buses, October 18, 1999 (approval pending).

[3] IEEE Std 1394-1995, Standard for a High Performance Serial Bus.

[4] IEC 61883 Parts 1 – 5, Standard for a Consumer-Use Digital Interface.

[5] CDR: Object Management Group (OMG) CORBA specification 2.41.

[6] The Java Virtual Machine, Tim Lindholm and Frank Yellin, Addison-Wesley, 1997.

[7] The Java Language Specification, James Gosling, Bill Joy, and Guy Steele, Addison-Wesley, 1996.

[8] Java Development Kit (JDK) 1.1 Core API Specification (http://java.sun.com/products/jdk/1.1/docs/api/packages.html).

[9] PersonalJava 1.1 API Specification (http://java.sun.com/products/personaljava/spec-1-1/pJavaSpec.html).

[10] Portable Network Graphics (PNG) specification v1.0, IETF RFC 2083.

[11] Audio Interchange File Format, version C, allowing for Compression (AIFF-C), Digital Audio Visual Council (DAVIC) 1.3 Part 9, Annex B.

[12] UNICODE Standard Version 2.0.

[13] HAVi Test Requirements, The HAVi, Inc., January 2000.

[14] IEEE std 1394a-2000, Standard for a High Performance Serial Bus – Amendment I

[15] TA Document 1999032: Clarification and Implementation Guideline for Isochronous Connection Management of IEC 61883-1.

[16] HAVi Certification Procedures, The HAVi, Inc. [To be issued]

[17] HAVi Logo Requirements, HAVi, Inc. [To be issued]

[18] JPEG International Standard ISO 10918-1 using JPEG File Interchange Format (JFIF) Version 1.02.

[19] RSA algorithm for digital signature PKCS#1 v2.0, IETF RFC 2437.

[20] SHA-1 Message Digest, specified by National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)." FIPS Publication 180-1, April 17, 1995.

1.3 Terminology

HAVi Acronyms

Acronym	Description
BAV	Base AV device
DCM	Device Control Module
DDI	Data Driven Interaction
FAV	Full AV device
FCM	Functional Component Module
HJA	HAVi Java API
HUID	HAVi Unique Identifier

IAV	Intermediate AV device
LAV	Legacy AV device
SDD	Self Describing Device
SEID	Software Element Identifier

Other Acronyms

Acronym	Description
1394, IEEE 1394	IEEE std 1394-1995
API	Application Programming Interface
AV/C-CTS	Audio/Video Control Command and Transaction Set (specified by the 1394 Trade Association)
AV/C	Audio/Video Control
CDR	Common Data Representation [5]
CE	Consumer Electronics
CTS	Command and Transaction Set
DTV	Digital TV
DV	Digital Video, the consumer version of DVC
DVC	Digital Video Cassette
DVD	Digital Video/Versatile Disc
EPG	Electronic Program Guide
FCP	Function Control Protocol (IEC 61883.1)
GUI	Graphical User Interface
GUID	Global Unique Identifier
IR	infrared
iPCR	input Plug Control Register (IEC 61883.1)
oPCR	output Plug Control Register (IEC 61883.1)
PCR	Plug Control Register (IEC 61883.1)
RTOS	Real-time Operating System
STB	Set-top Box
bslbf	Bit string leftmost bit first
uimsbf	unsigned integer, most significant bit first
UI	User Interface

Definitions

Term	Description
Application Module	An application, of a form defined by HAVi, that may provide a DDI interface and/or means for obtaining havlets.
Application Module code unit	A code unit from which an Application Module is obtained.

attachment	A “stage” of an external connection involving resources within a single device.
base AV device (BAV)	A HAVi-compliant device containing SDD data but not running any of the software elements of the HAVi Architecture.
bytecode	Bytecode for the Java™ virtual machine (“Java bytecode”).
client	A software element controlling one or more resources (see also *device resource* and *network resource*).
code unit	Refers to the executable code from which HAVi software elements are obtained. *Native code units* are platform dependent and typically would include machine code for a specific processor. *Java code units* include Java bytecode and so are platform independent. *Embedded code units* are those that are pre-loaded, while *uploadable code units* are those that may be dynamically loaded (and unloaded).
connection	A unidirectional data transfer path created by a HAVi Stream Manager. Typically used for streaming content. Connections originate and/or terminate at functional components. A connection is either an *internal connection* or an *external connection*. *Non-HAVi connections* refer to data transfer paths created by non-HAVi applications or devices.
controller	A device which controls other devices. An IAV or FAV device.
Data Driven Interaction (DDI)	A HAVi mechanism allowing control of software elements (DDI Targets) with access to devices by other software elements (DDI Controllers) with access to user input/output facilities. In this interaction, the user interface is described by a set of data structures (DDI elements) sent between the DDI Controller and the DDI Target.
DDI Controller	A software entity which renders DDI elements and handles user interaction using its (typically local) input/output facilities.
DDI element	The DDI encoding of a user interface element. For example, buttons, icons, sliders, text displays and text entry fields.
DDI protocol	The HAVi messages supporting Data Driven Interaction.
DDI Target	A software entity (typically a DCM) which supports the DDI protocol thereby allowing an associated DDI Controller to use the target’s DDI elements to control a device associated with the DDI Target.
device	A physical entity attached to the home network, examples are video players/recorders, cameras, CD and DVD players, set-top boxes, DTV receivers, and PCs.
device connection	An internal connection or an attachment.
device control module (DCM)	A HAVi software element providing an interface for controlling general functions of a device.
device resource	An FCM.
DCM code unit	A code unit from which a DCM is obtained. Installation of a DCM code unit results in one DCM and zero or more FCMs.
embedded code unit	See code unit.
embedded DCM	A DCM pre-loaded on an FAV or IAV. Embedded DCMs typically run on IAV devices and are typically implemented in native code.

external connection	A connection where data is transferred across a device boundary.
full AV device (FAV)	A HAVi-compliant device which runs the software elements of the HAVi Architecture including a Java runtime environment.
functional component	An abstraction within the HAVi Architecture that represents a group of related functions associated with a device. For example a DTV receiver may consist of several functional components: tuner, decoder, audio amplifier etc.
functional component module (FCM)	A HAVi software element providing an interface for controlling a specific functional component of a device.
global unique ID (GUID)	A 64-bit quantity used to uniquely identify an IEEE 1394 device. Consists of a 24-bit vendor ID (obtained from the 1394 Registration Authority Committee) and a 40-bit serial number assigned by the device manufacturer. The GUID is stored in a device's configuration ROM and is persistent over 1394 bus resets.
HAVi Architecture	The HAVi Architecture comprises the messaging model, control model, device model, and execution environment defined in this document.
HAVi-compliant device	A device conforming to the HAVi Architecture specification for an FAV, IAV or BAV device.
HAVi Java API	The Java API is defined in this specification.
HAVi Level 1 interoperability	Refers to the features provided by IAVs and embedded DCMs.
HAVi Level 2 interoperability	Refers to the features provided by FAVs and uploaded DCMs.
HAVi RMI	HAVi defined RMI (Remote Method Invocation) based on request and response exchanges among HAVi Messaging Systems.
HAVi unique ID (HUID)	A unique identification of devices and their functional components. Persistent over changes in network configuration (i.e., due to device plug-in or plug-out).
havlet	A HAVi Java application that is uploaded on the request of a controller from a DCM or Application Module.
havlet code unit	A code unit from which a havlet is obtained.
home network	The home network is the generic name used to define the communications infrastructure within the home. This name is used as an abstraction from the physical media and associated protocols. A home network supports both the exchange of control information and the exchange of AV content.
intermediate AV device (IAV)	A HAVi-compliant device which runs the software elements of the HAVi Architecture but does not include a Java runtime environment.
internal connection	A connection where data is transferred within a device.
Java code unit	See code unit.
legacy AV device (LAV)	A non HAVi-compliant device.
native code unit	See code unit.
network resource	IEEE 1394 bandwidth or an IEEE 1394 channel.
scheduled action	A set of operations, involving devices on the home network,

	to be performed at a specific time. For example, tuning to and recording a television program.
SDD data	Self Describing Device (SDD) data is stored in the IEEE 1212 Configuration ROM found on 1394 devices. HAVi specifies Configuration ROM data items that may be used for uploaded DCMs in the HAVi-specified format or for DDI elements in a vendor-specific format.
software element	A HAVi object. A software element responds to a set of messages specified by the API for that element.
software element ID (SEID)	An 80-bit value used to identify software elements. Not guaranteed to be persistent over changes in network configuration (i.e., due to device plug-in or plug-out).
system component	A software element providing basic system services. The system components are: 1394 Communication Media Manager, Messaging System, Event Manager, Registry, DCM Manager, Stream Manager and Resource Manager.
system software element	See system component.
uploaded / uploadable code unit	See code unit.
uploaded / uploadable DCM	A DCM dynamically loaded on an FAV or IAV. HAVi provides support for installing uploadable DCMs implemented in Java bytecode. Installing other forms of uploadable DCMs is vendor dependent.

1.4 Compliance

Each HAVi compliant device (FAV, IAV and BAV) shall:

- support the IEEE1394-1995 and the IEEE1394a-2000 amendment specification.
- provide HAVi SDD data in a IEEE 1212 configuration ROM.
- if the device sources or sinks a stream type for which IEC 61883 transmission has been specified, then the device should support: the PCR and CMP rules for isochronous connections as defined in IEC 61883.1, the CIP protocol as defined in IEC 61883.1, the CIP format specific definition in the corresponding part of IEC 61883.

A HAVi compliant BAV device shall:

- comply to the general HAVi compliance rules described above.
- contain in its HAVi SDD either a signed Java code unit plus corresponding profile, or a reference to a URL containing a signed Java code unit plus corresponding profile.

A HAVi compliant IAV device shall:

- comply to the general HAVi compliance rules described above.
- support IEC 61883.1
- run the following HAVi system components: 1394 Communication Media Manager, Messaging System, Event Manager, and Registry.
- run a DCM Manager if it can host DCMs for LAV or BAV devices.
- run a Stream Manager if software elements on the device require the establishment of streaming connections.
- run a Resource Manager if the device hosts or can host DCMs.
- generate a bus reset if it goes into a standby or low power mode for which the HAVi system is no longer running.
- provide persistent memory for use by HAVi software elements if DCM hosting is supported.

A HAVi compliant FAV device shall:

- comply to the general HAVi compliance rules described above.
- support IEC 61883.1
- run the following HAVi system components: 1394 Communication Media Manager, Messaging System, Event Manager, Registry, DCM Manager, Stream Manager and Resource Manager.
- generate a bus reset if it goes into a standby or low power mode for which the HAVi system is no longer running.
- provide persistent memory for use by HAVi software elements.
- run a Java runtime environment.
- implement the HAVi Java APIs as listed in Chapter 7.
- run a DCM representing itself.

A HAVi compliant bytecode DCM shall:

- refer only to classes specified in HAVi FAV profile #1: *DCM and Application Modules*, or implemented by the DCM itself.

A HAVi compliant bytecode Application Module shall:

- refer only to classes specified in HAVi FAV profile #1: *DCM and Application Modules*, or implemented by the Application Module itself.

A HAVi compliant havlet shall:

- refer only to classes specified in HAVi FAV profile #2: *Havlets*, or implemented by the havlet itself.

A set of test requirements for HAVi devices can be found in [13].

2 Overview

The HAVi Architecture specifies a set of Application Programming Interfaces (APIs) allowing consumer electronics manufacturers and third parties to develop applications for the home network. Thus the home network is viewed as a distributed computing platform, and the primary goal of the HAVi Architecture is to assure that products from different vendors can *interoperate* – i.e., can cooperate to perform application tasks.

To explain fully the interoperability aspects of the architecture, it is necessary to begin with an overview of home networking and identify the requirements addressed by the HAVi Architecture.

2.1 The Home Network

Current CE devices, such as DVD players and DV camcorders, are sophisticated digital processing and digital storage systems. By connecting these devices in networks, it is possible to share processing and storage resources – this allows new applications that:

- coordinate the control of several CE devices simultaneously, and
- simplify operation of devices by the user.

For instance one device may initiate recording on a second while accessing EPG (Electronic Program Guide) information on a third. The home network provides the fabric for connecting CE devices. It allows connected devices to exchange both control information (one device sending a command to another) and AV content (one device sending an audio or video stream to another). To be successful in the consumer electronics domain the home network must meet several requirements. These include: timely transfer of high-data-rate AV streams, self-configuration and self-management, hot plug-and-play, and low-cost cabling and interfaces. The HAVi Architecture is intended for networks based on the IEEE 1394 standard. 1394 is a powerful technology that meets many of the requirements of home networks. An example of a 1394 network is shown below:

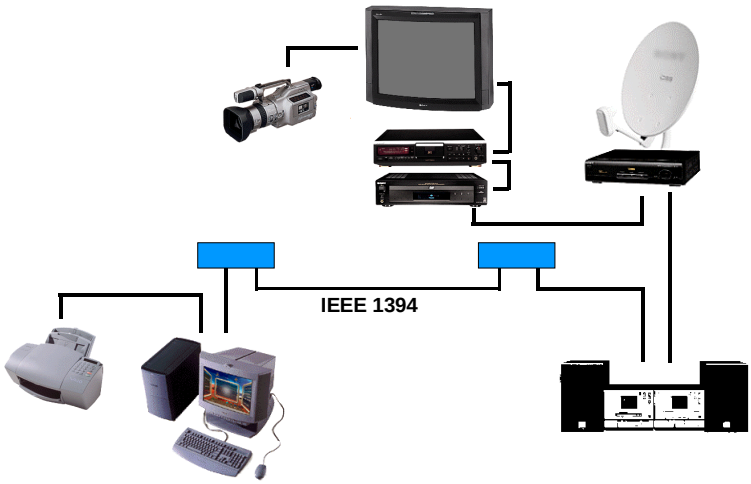

Figure 1. A 1394 Network with AV Clusters

The underlying structure for the home network consists of interconnected clusters of devices. Typically, there will be several clusters in the home, with one per floor or one per room. Each cluster will work as a set of interconnected devices to provide services to users. Often one device will control other devices. However, the HAVi Architecture is sufficiently flexible to allow home networks with no single master control device.

2.2 Requirements

The HAVi Architecture is an open, light-weight, platform-independent and architecturally neutral specification that allows consumer electronics manufacturers to develop interoperable devices, and independent application developers to write applications for these devices. It can be implemented on different hardware/software platforms and does not include features that are unique to any one platform.

The interoperability interfaces of the HAVi Architecture are extensible and can be advanced as market requirements and technology change. They provide the infrastructure to control the routing and processing of isochronous and time-sensitive data such as audio and video content.

2.2.1 Legacy Device Support

The HAVi Architecture supports legacy devices, i.e., devices that already exist and are available to users. This is important since the transition to networked devices is going to be gradual – with manufacturers not suddenly producing only networked devices and consumers not suddenly replacing their existing devices.

Legacy devices can also be characterized by the degree to which they support 1394 and industry standard protocols for 1394 such as IEC 61883. In particular, legacy devices can be divided into the following categories:

- non-1394 devices
- 1394 devices not supporting the HAVi Architecture

Most existing CE devices fall into the first category, while existing devices with 1394 interfaces fall into the second category.

HAVi-compliant devices, as opposed to legacy devices, are those that support the HAVi Architecture. The various categories of HAVi-compliant devices are described in section 2.3.3.

2.2.2 Future-Proof Support

The CE industry has great concern that new products work with existing products. While currently this is largely a question of media formats and interconnect standards, the HAVi Architecture supports future devices and protocols through several software-based mechanisms. These include:

- persistent device-resident information describing capabilities of devices
- a write-once, run-everywhere language (*Java*), used for software extensions
- a device independent representation of user interface elements

Each HAVi-compliant device may contain persistent data concerning its user interface and device control capabilities. This information can include Java bytecode that can be uploaded and executed by other devices on the home network. As manufacturers introduce new models with new features they can modify the bytecode shipped with the device. The new functionality added to the bytecode mirrors the new features provided by the device. Similarly new user interface elements can be added to the stored UI representation on the device.

2.2.3 Plug-and-Play Support

Home network consumer devices are easy to install, and provide a significant portion of their value to the consumer without any action on the user's part, beyond physically connecting the cables. This is in distinction to existing devices that require configuration to provide some major portion of their functionality. Home networking technology offers "hot" plug-and-play (not requiring the user to switch off devices), and safe and reliable connections.

In the HAVi Architecture, a device configures itself, and integrates itself into the home network, without user intervention. Low-level communication services provide notification when a new device is identified on the network.

While there will often be settings the user may change to suit his or her preferences, the HAVi Architecture does not require the user to perform any additional installation operations (as compared to installation for non-networked or stand-alone usage). Frequently, installing a device on the home network will be simpler than stand-alone installation since new devices can obtain configuration information from those already on the network. Thus the infamous "flashing clock on the VCR" can be solved by having the VCR set its clock to that of another device on the network – for example a DTV receiving time signals via digital broadcast.

2.2.4 Flexibility

The HAVi Architecture allows devices to present multiple user interfaces, adapting to both the

user's needs and the manufacturer's need for brand differentiation. The architecture includes a flexible device model that scales gracefully from simple CE devices like a CD player or audio amplifier to resource-rich, intelligent devices such as DTV receivers.

2.3 System Model

2.3.1 Control Model

The home network is considered to consist of a set of AV devices. Each device has, as a minimum, enough functionality to allow it to communicate with other devices in the system (with the exception of legacy devices, see section 2.3.3).

During the course of interaction, devices may exchange control information and data in a peer-to-peer fashion. This ensures that, at the communication level, no one device is required to act as a master or controller for the system. However, it also allows a logical master or controller to impose a control structure on the basic peer-to-peer communication model.

The HAVi control model makes a distinction between *controllers* and *controlled devices*. A controller is a device that acts as a host for a controlled device. A controlled device and its controller may reside on the same physical device or on separate physical devices.

In terms of the HAVi control model, a controller is said to host a *Device Control Module* (DCM) for the controlled device. The control interface is exposed via the API of this DCM. This API is the only access point for applications to control the device.

For instance, an intelligent television in the family room might be the controller for a number of interconnected devices. A controlled device could contain Java bytecode that constructs a user interface for the device and allows external control of the device. When these devices are first connected, the controller obtains the user interface and control code. An icon representing the device may then appear on the television screen, and manipulating the icon may cause elements of the control program to actuate the represented device or devices in prescribed ways.

The home network allows a single device, or a group of devices communicating amongst themselves, to deliver a service to a user or an application. When it is necessary for a device to interact with a user, a GUI for the device may be presented on a device with display capabilities (possibly the device in question or possibly a different device).

DCMs are a central concept to the HAVi architecture and the source of flexibility in accommodating new devices and features. DCMs can be distinguished in several ways (see the figure below).

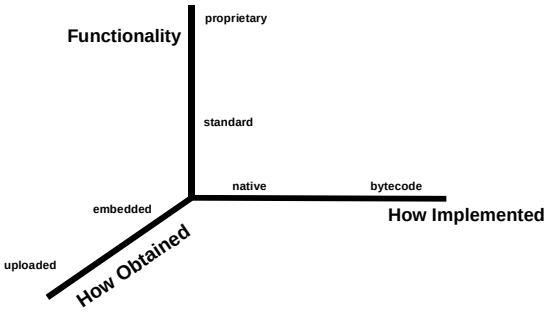

Figure 2. DCM Characteristics

The first DCM characteristic is how the DCM is obtained by the controller:

- *embedded DCM* – a DCM that is part of the resident software on a controller.
- *uploaded DCM* – a DCM that is obtained from some source external to the controller and dynamically added to the software on the controller.

The second characteristic is whether a DCM is platform (controller) dependent or platform independent:

- *native DCM* – a DCM that is implemented for a specific platform, it may include machine code for a specific processor or access platform specific APIs.
- *bytecode DCM* – a DCM that is implemented in Java bytecode.

Finally, DCMs can be distinguished by their functionality (or, conversely, their range of use):

- *standard DCM* – a DCM that provides the standard HAVi APIs. Such a DCM provides basic functionality but is able to control a wide range of devices.
- *proprietary DCM* – a DCM that provides vendor-specific APIs (in addition to the standard HAVi APIs). Such a DCM would offer additional features and capabilities over a standard DCM but could control a narrower range of devices, perhaps only a specific device or model.

HAVi provides support for uploaded DCMs written in Java bytecode so, in the remainder of this document, “uploadable DCM” implies a bytecode DCM unless indicated otherwise. (Note – native uploaded DCMs are possible but are beyond the scope of this document since their installation requires vendor-specific extensions to HAVi.)

2.3.2 Device Model

A distinction is made between *devices* and *functional components*. A good example of this distinction can be found in a normal TV set. Although the TV set is generally one physical box, it contains several distinct controllable entities, e.g. the tuner, display, audio amplifier, etc. The controllable entities within a device are called functional components.

2.3.3 Device Classification

HAVi classifies CE devices into four categories: Full AV devices (FAV), Intermediate AV devices (IAV), Base AV devices (BAV), and Legacy AV devices (LAV). HAVi-compliant devices are those in the first three categories, all other CE devices fall into the fourth category. Referring to the distinction between controllers and controlled devices – FAVs and IAVs are controllers while BAVs and LAVs are controlled devices. The presentation associated with products of the various categories in the marketplace is defined in the HAVi Logo Requirements document [17].

2.3.3.1 Full AV Devices

A Full AV device contains a complete set of the software elements comprising the HAVi Architecture (see section 2.4.4). This device class generally has a rich set of resources and is capable of supporting a complex software environment. The primary distinguishing feature of an FAV is the presence of a runtime environment for Java bytecode. This allows an FAV to upload bytecode from other devices and so provide enhanced capabilities for their control. Likely candidates for FAV devices would be Set Top Boxes (STB), Digital TV receivers (DTV), general purpose home control devices, and even Home PC's.

2.3.3.2 Intermediate AV Devices

Intermediate AV devices are generally lower in cost than FAV devices and more limited in resources. They do not provide a runtime environment for Java bytecode and so cannot act as controllers for arbitrary devices within the home network. However an IAV may provide native support for control of particular devices on the home network.

2.3.3.3 Base AV Devices

These are devices that, for business or resource reasons, choose to implement future-proof behavior by providing uploadable Java bytecode, but do not host any of the software elements of the HAVi Architecture. These devices can be controlled by an FAV device via the uploadable bytecode or from an IAV device via native code. The protocol between the BAV and its controller may or may not be proprietary. Communication between a FAV or IAV device and a BAV device requires that HAVi commands be translated to and from the command protocol used by the BAV device.

2.3.3.4 Legacy AV Devices

LAV devices are devices that are not aware of the HAVi Architecture. These devices use proprietary protocols for their control, and quite frequently have simple control-only protocols. Such devices can work in the home network but require that FAV or IAV devices act as a gateway. Communication between a FAV or IAV device and legacy device requires that HAVi commands be translated to and from the legacy command protocol.

2.4 HAVi Software Architecture

2.4.1 Object-Based

Services in the HAVi Architecture are modeled as objects. Each object is a self-contained entity, called a *software element*, accessible through a well-defined interface and executing within a software execution environment hosted by the device on which the object runs. Note that different

devices may host different execution environments. Services are accessed, using the communications infrastructure, via their well-defined interfaces.

Services in the HAVi Architecture can be provided by device manufacturers, or can be added by third party vendors. The software model makes no distinction between "standard" services and vendor services; they are both implemented as objects.

2.4.2 Software Element Identifiers

Each object is uniquely named. No distinction is made between objects used to build system services and those used for application services. Objects make themselves known via a system wide naming service known as the *Registry*.

Objects in the system can query the Registry to find other objects and can use the result of that query to send messages to those objects.

The identifier assigned to an object is created by the *Messaging System* before an object registers. These identifiers are referred to as SEIDs – *Software Element Identifiers*. SEIDs are guaranteed to be unique, however the SEID assigned to an object may change as a result of reconfiguration of the home network (for example, device plug-in or removal, or re-initialization of a HAVi device).

2.4.3 Message-Based Communication

All objects communicate using a message passing model. Any object that wishes to use the service of another object does so by using a general purpose message passing mechanism that delivers the service request to the target object. The target object is specified using the unique SEID discussed above.

This general purpose message passing mechanism abstracts from the details of physical location, i.e. there is no distinction between an object on the same device and one on a remote device. The actual implementation of the message passing mechanism will differ from device to device and between vendors. However, the format of HAVi messages, and the protocol used for their delivery, must be common so that interoperability is assured.

The general intent of the object model and Messaging System is to provide a completely generic software model that is sufficiently flexible to allow multiple implementations with a variety of software systems and languages. Details of the binding between messages and the code that handles them are left to the system implementor.

2.4.4 Software Elements

The software elements of the HAVi Architecture support the basic notions of network management, device abstraction, inter-device communication, and device user interface (UI) management. Collectively these software elements expose the *Interoperability API*, a set of services for building portable distributed applications on the home network. The software elements themselves reside above a vendor specific platform such as a real-time operating system (RTOS). The diagram below depicts the arrangement of software elements on an FAV device. While not intended as an implementation blueprint, the diagram does highlight how the HAVi software elements form a middle layer between platform specific APIs and platform independent applications.

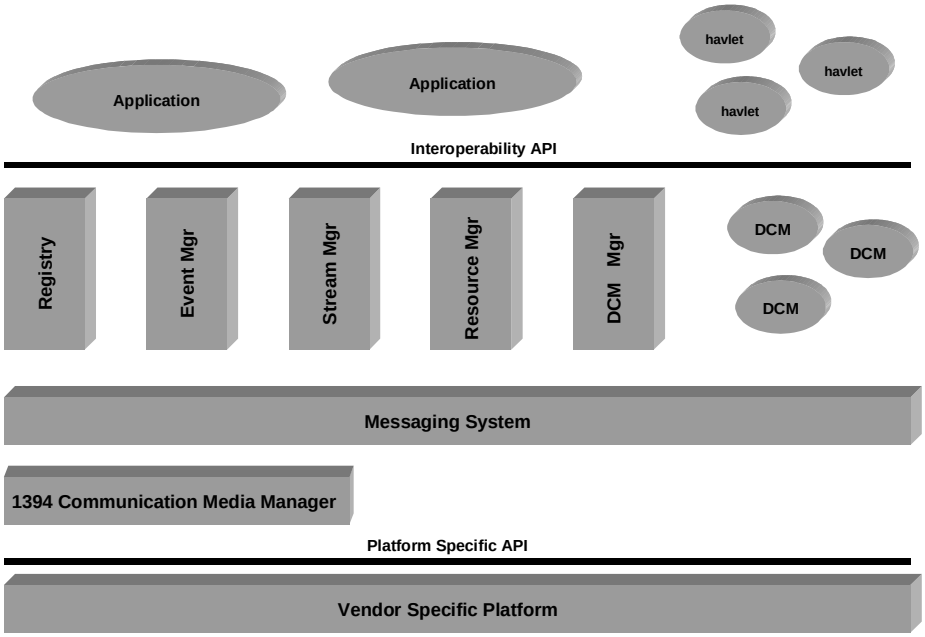

Figure 3. HAVi Architectural Diagram (FAV)

The software elements comprising the HAVi Architecture and defined in this specification are:

- *1394 Communication Media Manager* – allows other software elements to perform asynchronous and isochronous communication over 1394.
- *Messaging System* – responsible for passing messages between software elements.
- *Registry* – serves as a directory service, allows any object to locate another object on the home network.
- *Event Manager* – serves as an event delivery service. An event is the change in state of an object or of the home network.
- *Stream Manager* – responsible for managing real-time transfer of AV and other media between functional components.
- *Resource Manager* – facilitates sharing of resources and scheduling of actions.
- *Device Control Module* (DCM) – a software element used to control a device. DCMs are obtained from *DCM code units*. Within a DCM code unit are code for the DCM itself plus code for *Functional Component Modules* (FCMs) for each functional component within the device. In addition a DCM code unit may include a *havlet* allowing user control of the device and its functional components.
- *DCM Manager* – responsible for installing and removing DCM code units on FAV and IAV devices.

In addition to the above software elements specified by the HAVi Architecture, devices on the home network may contain the following:

- *Application Module* – The HAVi architecture provides a general platform for several forms of applications. In general a *HAVi application* is one that creates software elements that use other software elements to provide specific services. A HAVi application may be developed in native code and embedded (resident) on an FAV or IAV. It is also possible for a HAVi application to be in the form of Java bytecode and obtained from external sources (i.e., uploaded over the Internet) or from existing software elements using mechanisms defined by HAVi. In particular, an Application Module is a software element that may provide a DDI interface and/or a havlet.

- *Self Describing Device (SDD) data* – HAVi-compliant devices contain descriptive information about the device and its capabilities. This information, called SDD data, follows the IEEE 1212 addressing scheme used for Configuration ROM. The SDD data may include a DCM code unit and vendor-specific data for constructing user interface elements.

- *Java Runtime Environment* – provides an execution environment for uploaded DCMs and applications implemented using Java bytecode.

- *DDI Controller* – a software element involved with user interaction. The DDI (Data Driven Interaction) Controller handles user input and interprets (renders) DDI elements.

The following table summarizes which architectural elements are present for the various device categories, which are absent and which are optional. An optional element is indicated by "[]", a "check mark" indicates that the element is present on the device itself with the following provisions:

- A DDI Controller is required on an IAV or FAV if the device will render DDI elements according to the DDI protocol. Display-capable IAVs or FAVs must contain DDI Controllers.

- A Resource Manager is required on an IAV device that hosts or can host DCMs.

- A Stream Manager is required on an IAV if applications on the IAV will make streaming connections.

- A DCM Manager is required on an IAV device if it can host DCMs for BAVs or LAVs on the 1394 network.

- For both BAV and LAV devices there must be an associated device control module somewhere on the home network. For a BAV device, the DCM is typically obtained via the device's SDD data. For LAV devices, the DCM may be embedded on the controlling FAV or IAV.

- For IAV devices it is not necessary that a DCM exists on the home network. (However, if such a DCM does not exist then interoperable applications cannot control the device.)

- A DCM for an FAV or IAV resides on the device itself – i.e., an IAV (if it has a DCM) or an FAV hosts its own DCM.

Table 1. HAVi Configurations

Device Class / Element	FAV	IAV	BAV	LAV
Java Runtime	✔			
Application Module	[✔]	[✔]		
DDI Controller	[✔]	[✔]		
Resource Manager	✔	[✔]		
Stream Manager	✔	[✔]		
DCM Manager	✔	[✔]		
Registry	✔	✔		
Event Manager	✔	✔		
Messaging System	✔	✔		
1394 Communication Media Manager	✔	✔		
SDD data	✔	✔	✔	
DCM	✔	[✔]	✔	✔

2.5 User Interface Support

The primary goal of the user interface of the home network is to offer users an easy to use operating environment. The HAVi Architecture allows users to control devices through familiar means, such as via the front panel or via the buttons of a remote controller. In addition the HAVi Architecture allows device manufacturers to specify graphical user interfaces (GUIs) which can be rendered on a range of displays varying from text-only to high-level graphical displays. The GUI need not appear on the device itself, it may be displayed on another device and the display device may potentially be from another manufacturer. To support this powerful feature, the HAVi Architecture provides two mechanisms – the first, the *Level 1 UI*, is intended for IAVs and is called *Data Driven Interaction* (DDI), the second, the *Level 2 UI*, consists of Java APIs that may be provided by FAVs.

2.5.1 Level 1 UI

In essence, SDD data may include, in a vendor dependent manner, *DDI elements* – a platform independent encoding of user interface elements. DDI elements can be loaded from a *DDI Target*, typically a DCM, and displayed by a *DDI Controller*. The DDI Controller retrieves the DDI elements via the DCM (rather than directly from the SDD data, so it is possible that the DCM itself is the source of DDI elements). The DDI Controller generates HAVi messages in response to user input, it also responds to HAVi messages sent by the DCM as a result of changes in device state. This communication is called the *DDI protocol*.

It should be emphasized that the DDI Controller does not understand what happens as a result of issuing or responding to a control message. The DDI protocol involves only abstractions of user interface elements and user actions and is independent of any particular device semantics. Therefore, it is possible for a DDI Controller to handle new device functions which were not known at the time of DDI Controller implementation.

The DDI Controller cannot provide guarantees over the graphical rendition of DDI elements actually presented to the user, since their representation may be changed due to lack of display screen space or other Controller resource limitations. (Furthermore, application software can create different representations, using the DDI elements as "hints".) However the DDI Controller is required to try to preserve the appearance of DDI elements subject to its rendering capabilities.

2.5.1.1 Layout Mechanism

Layout rules of DDI elements are based on geometric coordinates and use *x, y* values for each DDI element. DDI elements are arranged in a hierarchy and positioned relative to their parents. The top level of hierarchy is a DDI *panel*.

The DDI Target *suggests* a preferred layout, which is encoded into the DDI data structure. However, the DDI Controller may tailor the presentation of DDI elements based on its own limitations, such as screen size, ability to display graphics or text only, etc.

2.5.1.2 Navigation Mechanism

The navigation between DDI elements within the same panel is handled locally by the DDI Controller. The DDI Target may suggest navigation rules between certain DDI elements. Because such navigation rules are just suggestions, the controller may tailor the navigation of the display based on its adjustment of DDI layout.

2.5.2 Level 2 UI

A Level 2 user interface is constructed by bytecode applications running on FAVs. The Java APIs used for implementing a Level 2 UI are based on a subset of Java AWT 1.1 and the following extensions specified by HAVi:

- support for different pixel aspect ratios, screen aspect ratios and screen sizes
- support for alpha blending and video / image layering
- support for remote control input
- support for a set of visual interface components patterned after the features offered by the Level 1 DDI elements

2.5.3 User Notification

There are various events generated by HAVi software elements that must come to the attention of the users of the HAVi system in order that they may react to the situations signaled by these events. Examples include the events generated by the Resource Manager, Stream Manager or Registry. Furthermore, applications and devices on the home network may be presented visually to the user, and information presented to the user should accurately reflect the state of applications and devices. Consistent presentation of state is particularly important since: 1) a device may be acted on in several ways: via its front panel and via a Level 1, Level 2, or arbitrary application accessing its DCM; and 2) several users may simultaneously access or view the same device. For BAV and LAV devices an (icon) representation can be presented to the user. Through such a representation, a user may request to use or manipulate a device or its data.

In order to disseminate information on system state to all users, it is recommended that the notifications of HAVi events be presented (in a vendor dependent way) on all display-capable FAV and IAV controllers in the network. There may be (vendor dependent) facilities allowing the user to disable such notifications. Annex 11.9 lists the events defined by HAVi. Certain events, such as those generated when installing or uninstalling DCMs can be used to detect the addition or removal

of devices and so can help provide a global view of the home network. Other events can be used to relay to the user information concerning the operating state of devices. Still other events can be used to inform the user of anomalous conditions such as communication failure. And yet other events can be used to inform the user of the failure to complete a previously scheduled activity (such as a planned recording at a future date).

2.6 Home Network Configurations

The HAVi Architecture defines how devices are abstracted within the home network and establishes a framework for device control. It defines APIs and messaging protocols so that interoperability is assured, and it defines how future devices and services can be integrated into the architecture. The HAVi Architecture makes no restrictions, however, on what types of devices must be present in the home network. As a result several configurations are possible – networks without FAV devices, networks with multiple FAV devices, networks with LAV and BAV devices only, etc. Depending upon the types of devices on the home network, several different operational configurations are possible.

2.6.1 LAV and BAV Only

The HAVi Architecture does not provide any support for networks consisting of only BAV and LAV devices. However with the addition of a HAVi controller (an IAV or FAV) to the network, these devices can be made available to applications.

2.6.2 IAV or FAV as Controller

IAV and FAV devices act as controllers for the other device classes and provide a platform for the system services comprising the HAVi Architecture. To achieve this, FAVs may host Java bytecode DCMs while IAVs may host embedded DCMs. From an interoperability perspective, the primary role of a controller is to provide a runtime environment for DCMs. Applications use the APIs provided by the DCM to access the controlled device.

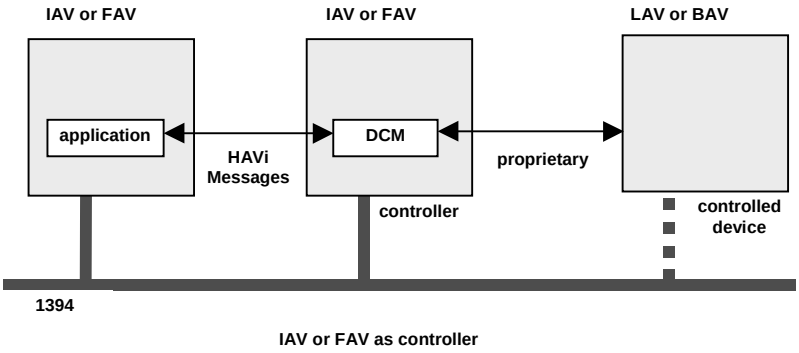

Figure 4. HAVi Controllers

2.6.3 IAV or FAV as Display

Generally, IAVs and FAVs will have an associated display device that is used for display of AV content and GUIs. However, the architecture does not mandate this and an IAV or FAV device

may be “headless” (without display capability). In this case they will cooperate with other IAV or FAV devices with display capability. A display capable IAV is required to support a DDI Controller. A display capable FAV is required to support a DDI Controller and a Level 2 UI. Proprietary low-level graphic manipulation APIs can be used by the DDI Controller to access the display itself, but these interfaces are not exposed as part of the Interoperability APIs.

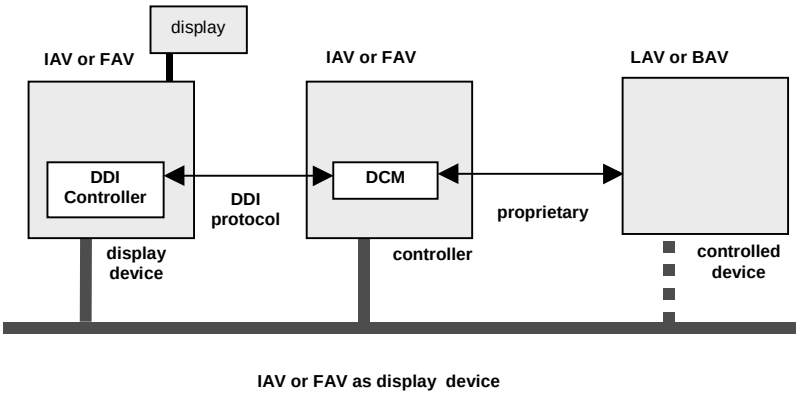

Figure 5. HAVi Displays

2.6.4 Peer-to-Peer Architecture between FAVs and IAVs

In a home network, there may be more than one FAV or more than one IAV. In this case, each controller (IAV or FAV) cooperates with other controllers to ensure that services are provided to the user. This allows devices to share resources. An example is described in section 2.6.3 where a device without display capabilities uses a remote device to display DCM user interfaces. A more elaborate example could be an FAV device utilizing the services of a data conversion module located on a remote device to set up a data stream between two AV devices.

2.6.5 IAV as Controller and Display

A home network may contain no FAV devices, in such cases IAVs are the only entities which may control other devices. Although not equipped with a runtime environment for uploaded DCMs, an IAV may be shipped with a set of *embedded* DCMs. Embedded DCMs can be implemented as native applications on the IAV device and can use native interfaces to access the IAV’s display and other resources. Embedded DCMs, like DCMs in general, appear in the Registry provided by the HAVi Architecture and can be accessed from other devices on the home network by sending messages over the Messaging System. Embedded DCMs, like DCMs in general, may support the DDI protocol and so may participate in providing a user interface for the controlled device.

2.7 Interoperability in the HAVi Architecture

The first and foremost goal of the HAVi Architecture is to support interoperability between AV equipment. This includes existing equipment and future equipment. Because of the need to support existing devices, and because of product cost considerations, the HAVi Architecture supports two levels of interoperability. These are referred to as Level 1 and Level 2 respectively.

The flexibility of choosing different levels of interoperability is essential in allowing vendors the freedom to design and build devices at all points on the cost/capability spectrum.

2.7.1 Level 1 Interoperability

Level 1 interoperability addresses the general need to allow existing devices to communicate. To achieve this, Level 1 interoperability defines and uses:

- a generic set of control messages (commands) that enable one device to talk to another device and

- a set of event messages that it should reasonably expect from the device.

To support this approach a basic set of mechanisms are required.

- *Device discovery*: each device in the home network needs a well-defined method that allows it to advertise its capabilities to others. The approach the HAVi Architecture has adopted is to utilize SDD data, required on all FAV, IAV and BAV devices. SDD data contains information about the device which can be accessed by other devices. The SDD data contains, as a minimum, enough information to allow instantiation of an embedded DCM. This results in registration of device capabilities with the HAVi Registry, allowing applications to infer the basic set of command messages that can be sent to the device.

- *Communication*: once an application has determined the capabilities of another device, then it needs to be able to access those capabilities. To achieve this requires a general communication facility allowing applications to issue requests to devices. This service is provided by the HAVi Messaging Systems and DCMs. The application sends HAVi messages to DCMs, the DCM then engages in proprietary communication with the device.

- *HAVi message set*: the last mechanism required to support Level 1 interoperability is a well defined set of messages that must be supported by all devices of a particular class. This ensures that a device can work with existing as well as future devices, irrespective of the manufacturer. The HAVi message set includes those messages used for the DDI protocol and so allows DCMs (and applications) to construct a UI on display-capable IAVs and FAVs.

These three basic mechanisms support a minimal level of interoperability. Since any application can query the Registry, any application can determine the message set supported by any DCM. Since any application has access to the Messaging System, any application can interact with the DCM of any device.

2.7.2 Level 2 Interoperability

As described in the previous section, Level 1 interoperability ensures that devices can interoperate at a basic level of functionality. However, a more extensible mechanism is also needed to allow a device to communicate to other devices any additional functionality not present in embedded DCMs. For example, embedded DCMs may not support all features of existing products and are unlikely to support future product categories. Level 2 interoperability provides this mechanism.

To support non-standard features of existing products and to support future products, the HAVi Architecture allows uploaded DCMs as an alternative to embedded DCMs. The uploaded DCM may replace an existing DCM on FAV devices. The HAVi Architecture makes no statement about the source of the uploaded DCM, but a likely technique is to place the uploaded DCM in the SDD data of the BAV device, and upload from the BAV to the FAV device when the BAV is attached to the home network. Because the HAVi Architecture is vendor neutral, it is necessary that the

uploaded DCM will work on a variety of FAV devices all with potentially different hardware architectures. To achieve this, uploaded DCMs are implemented in Java bytecode. The Java runtime environment on FAV devices supports the instantiation and execution of uploaded DCMs.

Once loaded and running within an FAV device, the DCM communicates with the LAV and BAV devices in the same manner as described above in section 2.7.1.

The efficiency of Level 2 interoperability appears when one considers resources needed to access device functionality. Level 2 allows a device to be controlled via an uploaded DCM that presents all the capabilities offered by the device. Whereas to achieve similar functionality in Level 1, this DCM would have to be embedded somewhere in the network. For example when a new device is added to a network, Level 1 requires that at least one other device contains a DCM suitable for the new device. In comparison, Level 2 only requires that one device provide a runtime environment for the uploaded DCM obtained from the new device.

The concept of uploading and executing bytecode also provides the possibility for applications called *havlets*. Havlets may be device specific, for example a device manufacturer can provide the user a way to control special features of a device without the need for standardizing all the features in HAVi. Havlets can be uploaded and installed by each FAV device on the network. Havlet uploading can be supplied by DCMs and Application Modules and can offer interaction with the user via Level 2 UI. Display-capable FAVs will allow a user to upload and execute the havlet of any DCM or Application Module, providing a havlet, in the home network.

2.8 Versioning

Each HAVi component must support the HAVi version control API. HAVi version control is intended to maintain interoperability of HAVi components as the specification evolves. Version control for individual manufacturer's products is outside of the scope of this API.

Versions are represented by major and minor numbers in the form `major.minor`. For a given release of the specification, every system component will be of the same version as that of the HAVi specification – regardless of whether that component's API was modified in the latest release. This simplifies the situation in which a given component's APIs are built up from different groups of APIs. The minor version number is intended to indicate small refinements of the HAVi specification. The intent of the major version number is to reflect important functional improvements in the overall HAVi architecture.

Given that this specification is Version 1.1, all HAVi components are currently defined to be at Version 1.1.

As the HAVi specification evolves, it is intended that no APIs will ever be updated or removed (see section 5.1.7). All changes will be achieved by adding new APIs. This ensures that older components can always request services from newer components successfully.

All system components on a device shall return a unified version number in response to `GetVersion` API calls, and the value shall be identical to **HAVi_Message_Version** in the SDD of the device. System components shall verify the version number of their peer system components on other devices, e.g., by reading the **HAVi_Message_Version** value from the devices' SDD (see section 9.10.3). Each system component shall operate at the highest common version number among all peer system components in the network.

The rules for version control of message passing protocols is somewhat different. Please refer to the Messaging System section 3.2.1.2.7 for details.

Client applications are encouraged to interoperate with older software element versions, though this is not explicitly required. However system elements, DCMs and FCMs as clients shall interoperate with all older software element versions. This is intended to ensure that interoperability will be achieved for the life of the HAVi specification. Note that all software elements support a version number retrieval method (see section 5.13).

2.9 Security

All software elements can in principle send messages and events to each other without any restriction. However to avoid that applications send, whether accidentally or deliberately, messages or events to system components that were intended to be used only by other system components, a protection mechanism is needed.

HAVi specifies, for each defined HAVi message and event, the kinds of software element that are allowed to use it. The protection mechanism simply implies that a system component will check whether the message sender (or event poster) is allowed to send this message (or event). This check is based on an inspection of the SEID of the message sender (event poster).

Protection of a device (and the home network) from hostile or flawed applications is the responsibility of the vendor of the device. Such protection is particularly crucial for FAV devices since HAVi specifies an open programming environment for FAVs and arbitrary bytecode applications may be introduced to FAVs (e.g., via Web download, broadcast download, or installation from hard media).

2.9.1 Access Levels

HAVi uses a two-level protection scheme. When a software element is created it is assigned an access level which is one of trusted or untrusted. When one software element sends a request to another software element the receiver decides whether or not to honor the request by examining the access level of the requester (and optionally other information associated with the request).

1) System software elements should be thoroughly tested by vendors. They are assigned the trusted access level and operate with the greatest set of privileges.
2) For non-system software elements that are pre-installed on an IAV or FAV, the vendor shall test the software element for HAVi compliance. Provided there are no failures, the software element can be assigned the trusted level.
3) For software elements that are dynamically installed on an IAV or FAV through an installation mechanism that is proprietary and not publicly exposed, the vendor shall implement a proprietary verification mechanism and only assign the trusted level to software elements obtained from secure sources.
4) For software elements that are dynamically installed on an FAV through a public installation mechanism, the vendor shall implement the signature verification mechanism defined by HAVi and only assign the trusted level to software elements obtained from correctly signed sources.
5) If an IAV or FAV has a proprietary mechanism for installing patches to system elements or replacing system elements, then it is recommended that the vendor implement a proprietary verification mechanism and only allow patching or replacement of system software elements from secure sources.

All DCMs and FCMs shall be trusted. Therefore untrusted DCM code units shall not be installed on FAVs or IAVs.

Since HAVi does not specify a dynamic acquisition and installation mechanism for Application

Modules, if an IAV or FAV has such a feature then the vendor is responsible to assure only Application Modules obtained from secure sources are given the trusted level, with a vendor-dependent verification mechanism.

As for havlets, they are extracted from DCMs or Application Modules through a public method. Thus, a havlet code unit shall be verified whether it is correctly signed in a HAVi-compliant manner before it is installed on an FAV. If the verification fails, the FAV shall not assign the trusted level to the havlet.

It is vendor-dependent whether or not an FAV or IAV allows to install Application Modules or havlets which failed in the security verification. However, even when installation of such untrusted software elements is allowed, it is recommended that such an FAV or IAV has a proprietary mechanism to assure that such an installation is only done with the user's responsibility.

2.9.2 Signature Verification

The procedure of security verification of code resources used on IAVs is vendor dependent. Also, even on FAVs, the procedure of security verification for Application Modules is vendor dependent. However the digital signature algorithm and certification procedures are specified by HAVi, for the verification of uploadable DCM code units and havlet code units.

The procedure of signature verification on FAVs is as follows:

- Uploadable DCMs, uploadable Application Modules and all havlets are obtained from "code units" – these are JAR files.
- As for uploadable DCMs, the JAR file shall be signed in the HAVi-compliant manner, and the signature is verified when the file is loaded into the Java runtime. If there is no signature, or if signature verification fails, the DCM shall not be installed.
- As for havlets, the JAR file may be signed in the HAVi-compliant manner. In that case the signature is verified when the file is loaded into the Java runtime. If signature verification succeeds, then all classes defined in the file are trusted. If there is no signature, or if signature verification fails, all classes defined in the file are untrusted.
- As for Application Modules, the JAR file may have any information for vendor-dependent verification. If the verification succeeds, then all classes defined in the file are trusted. Otherwise, including the case an FAV is not aware of the verification mechanism, all classes defined in the file are untrusted.
- Only havlets and Application Modules created from trusted classes are trusted.

The digital signature algorithm used for uploadable DCMs and havlets on FAVs, and associated key management infrastructure, are specified in section 3.10.

3 Software Element Descriptions

3.1 Communication Media Manager

The Communication Media Manager (CMM) is a network dependent entity in the HAVi Architecture. It interfaces with the underlying communication media to provide services to other HAVi components or application programs residing on the same device as the CMM. Each physical communication medium has its own CMM to serve the above purpose. This section concentrates only on the CMM for the 1394 bus.

Two types of services are provided by the CMM. One is to provide a transport mechanism to send requests to and receive indications from remote devices. The other is to abstract the network activities and present information to the HAVi system. The 1394 bus is a dynamically configurable network. After each bus reset, a device may have a completely different physical ID than it had before. If a HAVi component or an application has been communicating with a device in the network, it may want to continue the communication after a bus reset, though the device may have a different physical ID. To identify a device uniquely regardless of frequent bus resets, the Global Unique ID (GUID) is used by CMM and other HAVi entities. A GUID is a 64 bit number that is composed of 24 bits of node-vendor ID and a 40 bit number assigned by the vendor (these are described in references [2] and [3]). While a device's physical ID may change constantly, its GUID is permanent. The CMM makes device GUID information available for its clients.

One of the advanced features the 1394 bus provides to the HAVi system is its support for dynamic device actions such as hot plugging and unplugging. To fully support this up to the user level, HAVi system components or applications need to be aware of these network changes. The CMM works with the Event Manager to detect and announce such dynamic changes in network configuration. Since any topology change within the 1394 bus will cause a bus reset to occur, the CMM can detect topology changes and post an event to the Event Manager about these changes along with associated information.

Trusted software elements are allowed to use their local CMM to initiate communication with other devices on the 1394 network, using the `Cmm1394::Write`, `Cmm1394::Read` and `Cmm1394::Lock` APIs defined in section 5.2.2. These APIs would typically be used by DCMs and FCMs to control remote 1394 devices, i.e. BAV and LAV devices. Communication between IAVs and FAVs, devices which support HAVi messaging, is typically accomplished using the HAVi defined software elements or the Messaging System directly.

For a software element to accept 1394-level communication initiated by a remote device it must first enroll for indications by calling the `Cmm1394::EnrollIndication` API. This API enables the software element to be notified of any write, read or lock operation from a specific device in a specific address range. This notification is performed by the `Cmm1394Indication` client API. The software element can drop previously enrolled indications by calling the `Cmm1394::DropIndication` API.

The CMM is intended to allow maximum flexibility so that DCMs and FCMs can individually control remote 1394 devices with a wide variety of functions and protocols. On a single IAV or FAV there could be a number of different DCMs and FCMs, each one controlling different remote 1394 devices using various 1394 address ranges. Note that if different 1394 remote devices perform a read of a given address range, they may receive different results, since each read request may be processed by an independent DCM or FCM. Also note that the response to such a read request (or lock request) is provided by the `responseData` argument of the `Cmm1394Indication` message,

rather than by a simple memory mapping.

3.2 Messaging System

3.2.1 Description

The Messaging System provides HAVi software elements with communication facilities. It is independent of the network and transport layers. A Messaging System is embedded in all FAV and IAV devices. The Messaging System of a device is in charge of allocating identifiers (SEIDs) for the software elements of that device. These identifiers are first used by the software elements to register. They are then used by the software elements to identify each other within the home network: when a software element (A) wants to send messages to another software element (B) it has to use the software element identifier of B when invoking the Messaging System API.

The Messaging System is composed of the message layer and the *Transport Adaptation Module* (TAM) which provides a basic communication API that is medium dependent. The TAM uses the services of 1394 and IEC 61883 protocol layers to send and receive data on the network. It gathers features that are medium dependent. When the medium changes, the TAM has to change as well. The TAM is described in section 3.2.2.

3.2.1.1 Software Element Identifier Allocation

A software element identifier (SEID) is allocated by the Messaging System when requested by a software element. A software element must obtain a SEID if it wants to be registered on the home network, or if it wants to communicate (via HAVi messages) with other software elements.

The software element identifier is 10 bytes long.

Syntax	Number of bits	Identifier
SEID() {		
GUID	64	uimsbf
swHandle	16	uimsbf
}		

Figure 6. SEID Representation

`GUID` identifies a device within the home network. It is the 1394 EUI64 value that is available in the ROM of 1394 devices.

`swHandle` identifies a software element within one device. `swHandle` allocation is described in section 3.2.1.1.1.

`SEID`, being the concatenation of `GUID` and `swHandle`, identifies therefore a software element within the home network.

From the software element's point of view, the identifier is an atomic identifier that is 10 bytes long. Only the HAVi Messaging System and other HAVi system components are aware of the internal structure of software element identifiers.

3.2.1.1.1 Software Element Handle Allocation

`swHandle` is allocated by the Messaging System of a device when a software element requests a software element identifier. The Messaging System of a device is in charge of allocating unique handles to the local software elements. *It is recommended that the Messaging System avoids reuse of the handles if possible so as to avoid potential conflict* (see section 3.2.1.2).

3.2.1.1.2 Well-known Software Element Handles

There is a need to have well-known software element handles for system components. The handle values from 0x0 to 0x00ff are reserved for this purpose. These well-known software element handles are listed in Annex 11.4. To reach a system component the requester must know the SEID of the component. Usually it will use the Registry service to find SEIDs. Therefore the requester has to reach the local Registry API and, consequently has to know the local Registry SEID. To avoid such circularity, the Messaging System API provides a method to find the SEIDs of system components.

3.2.1.1.3 Trusted and Untrusted Software Element Handles

HAVi defines a set of APIs provided by system components and DCM/FCMs. Some of the API methods are protected in the sense that only a subset of software elements are authorized to access the method. Consequently software element are classified into two categories:

- trusted software elements (can access all HAVi APIs)
- untrusted software elements (can access only untrusted APIs)

Each API described in chapters 5 and 6 gives, for each API method, whether it can be used by trusted and untrusted software elements or just trusted software elements.

To identify whether a software element is trusted or not, two software element handle ranges are defined :

0x0000 to 0x7fff : trusted software elements

0x8000 to 0xffff : untrusted software elements

The well-known handles (i.e., system component handles – see Annex 11.4) are all in the trusted range. The Messaging System delivers a trusted SEID or an untrusted SEID to a requester according to the access level of this requester (see section 2.9).

3.2.1.2 Message Transfer Service

The Messaging System provides a connectionless data transfer service. Before being able to exchange any kind of information, a software element has to: 1) obtain a software element identifier (SEID), and 2) indicate a call back function to the Messaging System for receiving messages. This step is done via the `MsgOpen` function.

Once a software element has obtained an identifier, and should it want to send messages to another software element (known through its software element identifier), it must use the message transfer services of the Messaging System.

3.2.1.2.1 Message Transfer Supervision

When a software element wants to send data to another software element it may establish a *supervision* of that software element via the `MsgWatchOn` function. The purpose of having a supervision of a software element is to be told when a software element leaves the network. As long as a supervision is established, the Messaging System is in charge of detecting when a device (on which it has established a supervision) disappears from the network. If this occurs it shall close all its supervisions to the disappeared device by invoking the call back of the source software elements.

A supervision is always unidirectional: if the destination software element needs to be informed of the loss of the originating software element, then it must establish its own supervision of that software element.

After a supervision has been requested, the Messaging System monitors the existence of the "watched" object using the `Msg::Ping` facility described in section 5.3.3. The Messaging System also subscribes to events that allows it to detect: when the device hosting the destination software element goes down, when the destination software element leaves the network, when the remote device is initialized (reset), and when the remote HAVi system has entered an anomalous state..

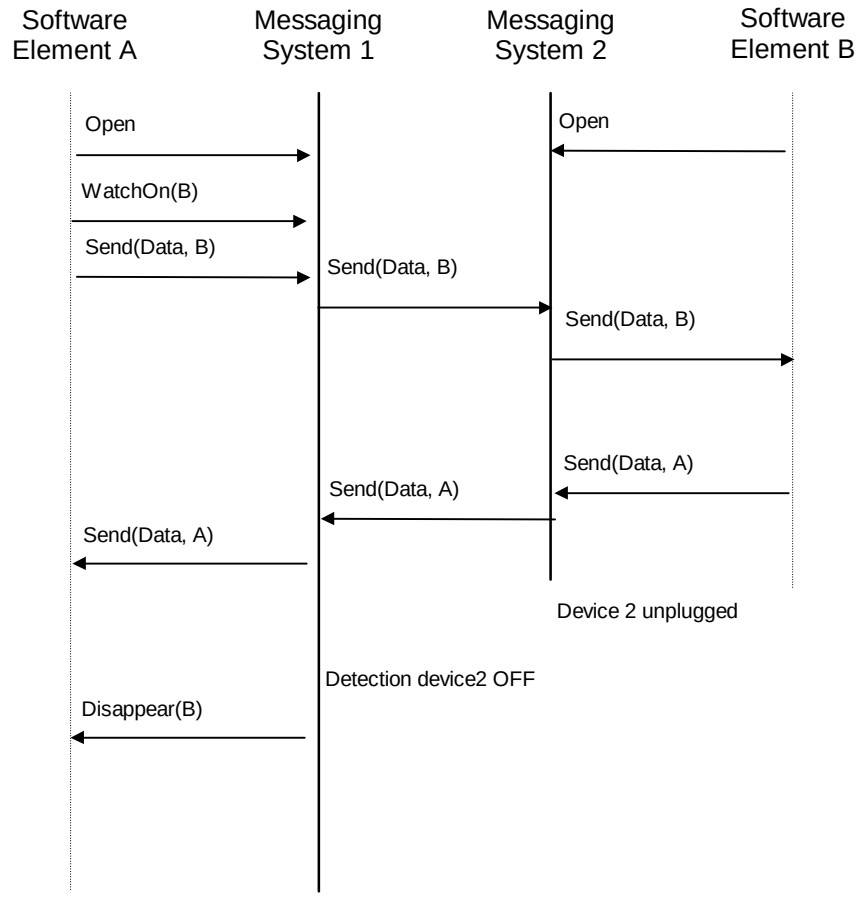

Figure 7. Example of Message Transfer Supervision

3.2.1.2.2 Message Transfer Modes

Data are not protected at the message level (i.e., there is no Messaging System CRC). However it is assumed that the lower layers provide an error detection service (via a CRC) as well as a good level of transmission reliability (see section 3.2.2). Therefore there are no error recovery mechanisms at

the message level (however the TAM provides an error recovery process according to 1394).

The Messaging System provides two modes to transmit a message: *simple mode* and *reliable mode*. The Messaging System API primitives `MsgSendReliable`, `MsgSendRequest` and `MsgSendRequestSync` use reliable mode, while `MsgSendSimple` uses simple mode and `MsgSendResponse` takes a `mode` parameter (see section 5.3.3).

Simple mode is very basic: no control is performed by the Messaging System. The message is sent on the network and that is all. This mode may be used when a software element wants to send a response to a request. The Messaging System does not check whether its response is received or not.

Reliable mode is more complicated and expects the destination device to acknowledge the message. The reliable mode is described through the following example: when a software element (A) running on a device D1 wants to send a reliable message to another software element (B) running on device D2, it invokes a Messaging System API with B's SEID as parameter. The Messaging System of D1 sends a `msg_reliable` message (see 3.2.1.2.4) to the Messaging System of D2. The Messaging System of D2 checks then whether it knows a call back for B. If yes, it invokes it. If the call back successfully returns (i.e., no error), it sends back a `msg_reliable_ack` to D1 (as an acknowledgement). If either no call back is available, or the call back returns with an error, D2's Messaging System sends back a `msg_reliable_noack` message which indicates an error. When the Messaging System of D1 receives the `msg_reliable_ack` (or `msg_reliable_noack`) message, the Messaging System API function invoked by A returns. Note that at the originating side, the calling software element is blocked until it gets the acknowledgement. To avoid blocking a software element indefinitely an acknowledgement timeout is used. Its value shall be 30 seconds. Expiry of the timer indicates to the Messaging System a major problem either locally or on the target. The timer is started at the end of transmission of the last TAM package of the message. In case device D1 detects that device D2 has disappeared before the `msg_reliable_ack` has been received, the Messaging System of D1 shall immediately return from `MsgSendReliable` with the error code `Msg::EACK`. In this situation no `MsgTimeout` event is generated.

3.2.1.2.3 Acknowledgements

The general message structure includes an 8-bit field called the *message number*.

The Messaging System maintains a message number counter for each source software element. When the Messaging System issues a request (reliable or not), the counter is incremented (modulo 256) and the new value is sent within the message. In case of a `msg_reliable` message, the Messaging System at the destination node will send back the same counter value within the `msg_reliable_ack` (or `msg_reliable_noack`) message.

The Messaging System checks each incoming `msg_reliable_ack` or `msg_reliable_noack`. If no reliable message of the destination software element with the same message number is pending, the incoming ack (or noack) is simply discarded.

In the case the response is received prior to the `msg_reliable_ack` of a request, the Messaging System may treat the response as a `msg_reliable_ack`. This may recover from a lost `msg_reliable_ack` in the case where the response is received before the acknowledgement timeout.

The Messaging System that resides on the same node as the software element receiving the request message shall send `msg_reliable_ack` and response message in order. Thus, the software element which receives a callback invocation has to immediately return from the callback.

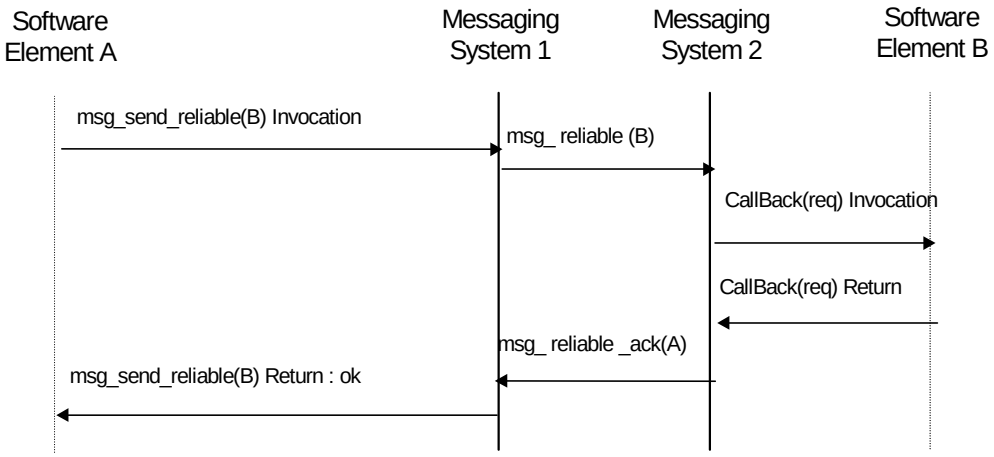

Figure 8. Typical Reliable Message Sequences

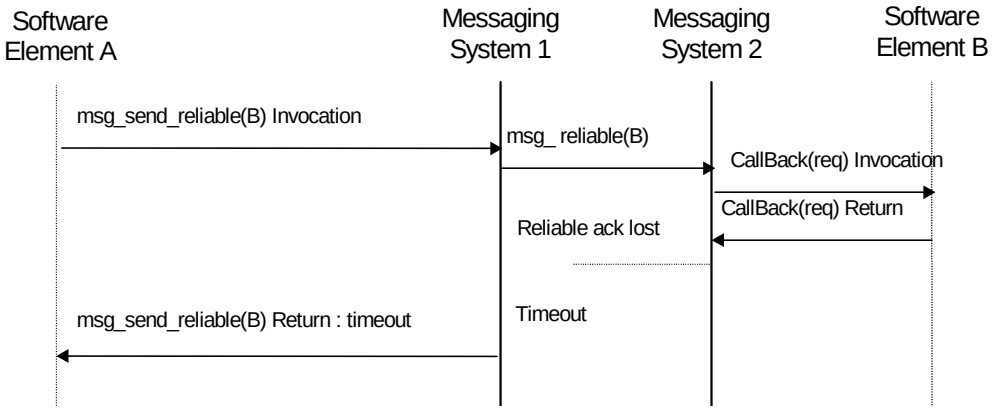

Figure 9. Reliable Messaging Failing Due to Timer Expiration

3.2.1.2.4 General Message Format

The message format shown in Figure 10 describes a message as it is sent by a Messaging System to another Messaging System.

Syntax	Number of bits	Identifier
message () {		
DestSEID	80	uimsbf
SourceSEID	80	uimsbf
ProtocolType	8	uismbf
MessageType	8	uimsbf
MessageNumber	8	uimsbf
reserved	8	uimsbf
MessageLength	32	uimsbf
for(j=0; j<MessageLength;j++){		
MessageBody[j]	8	uimsbf
}		
}		

Figure 10. General Message Format

`DestSEID` (10 bytes) is the identifier of the software element to which the message is to be sent.

`SourceSEID` (10 bytes) is the identifier of the software element that generated the message.

Note – When the message is a `msg_reliable_ack` or `msg_reliable_noack`, the `DestSEID` field will hold the `SourceSEID` of the related `msg_reliable` message.

`ProtocolType` (1 byte) is the format that `MessageBody` content must adhere to. The values from 0x00 to 0x7f are reserved for HAVi. The values from 0x80 to 0xff are free for private use. HAVi defines one particular protocol based on request and response message exchanges, providing applications with facilities to invoke operations on a given software element (request), and/or to return the result of an operation to the software element that requested it (response). The `ProtocolType` parameter in the general message format corresponding to the HAVi request/response mechanism has the value 0x00. For this value of `ProtocolType`, if `MessageType` (see below) is `msg_simple` or `msg_reliable` then the message body format must be as specified in section 3.2.3.2.

`MessageType` (1 byte) indicates the message type:

- One type is defined for the simple mode : the `msg_simple` message
- Three types are defined for the reliable mode: the `msg_reliable` message (carrying the request), the `msg_reliable_ack` (telling the request has been delivered), and the `msg_reliable_noack` (telling the request delivery failed).

`MessageNumber` (1 byte) is the message number. Its value is incremented according to the rules described in section 3.2.1.2.3.

`reserved` (1 byte), this field shall be set to zero.

`MessageLength` (4 bytes) is the length of the `MessageBody`. There is no payload for the `msg_reliable_ack`, in which case both the `MessageLength` and `MessageBody` fields are not

present.

MessageBody is the message data.

Table 2. Message Type Values

MessageType	Value
msg_simple	0x01
msg_reliable	0x02
msg_reliable_ack	0x03
msg_reliable_noack	0x04

3.2.1.2.5 Ack Message Format

The msg_reliable_ack message follows the general message format. It has no MessageLength and no MessageBody. The layout of an msg_reliable_ack message is indicated schematically below:

byte 0	byte 1	byte 2	byte 3	byte 4	...	byte 13
0011 rrrr	nnnn nnff	0Rnn nnnn	0000 0000	dddd dddd	...	dddd dddd
FCPHdr	TAMHdr		reserved	Destination SEID		

byte 14	...	byte 23	byte 24	byte 25	byte 26	byte 27
ssss ssss	...	ssss ssss	PPPP PPPP	0000 0011	NNNN NNNN	0000 0000
Source SEID			ProtType	MsgType	MsgNo	reserved

Destination SEID in the above is the SEID of the software element that sent the corresponding msg_reliable message. Source SEID is the SEID of the software element that received the msg_reliable message. The value of protocol type shall be the same as the value of the corresponding msg_reliable message.

3.2.1.2.6 Noack Message Format

The msg_reliable_noack message follows the general message format. The MessageBody contains one byte which may take the values shown below:

Table 3. msg_reliable_noack Message Body Values

Name	Value
SYSTEM OVERFLOW	0x01
UNKNOWN TARGET OBJECT	0x02
TARGET REJECT	0x03

SYSTEM OVERFLOW – memory allocation for the incoming message failed.

UNKNOWN TARGET OBJECT – the Messaging System cannot find a callback for the destination SEID embedded in the incoming message, or cannot deliver the response message for the synchronous function call to the caller software element.

TARGET REJECT – the return value of the callback is not equal to SUCCESS.

The layout of a `msg_reliable_noack` message is indicated schematically below:

byte 0	byte 1	byte 2	byte 3	byte 4	...	byte 13
0011 rrrr	nnnn nnff	0Rnn nnnn	0000 0000	dddd dddd	...	dddd dddd
FCPHdr	TAMHdr		reserved	Destination SEID		

byte 14	...	byte 23	byte 24	byte 25	byte 26	byte 27
ssss ssss	...	ssss ssss	PPPP PPPP	0000 0100	NNNN NNNN	0000 0000
Source SEID			ProtType	MsgType	MsgNo	reserved

byte 28	byte 29	byte 30	byte 31	byte 0
0000 0000	0000 0000	0000 0000	0000 0001	eeee eeee
Message Length				Message Body

Destination SEID in the above is the SEID of the software element that sent the corresponding `msg_reliable` message. Source SEID is the SEID of the software element that received the `msg_reliable` message. The value of protocol type shall be the same as the value of the corresponding `msg_reliable` message.

3.2.1.2.7 HAVi Message Version

The HAVi message version supported by a device is in its **HAVi_Message_Version** SDD field. Before the first exchange between two Messaging System modules, the initiator obtains the message version number from the receiver device. Then the initiator will use the message format defined in the highest HAVi specification shared by both Messaging Systems.

3.2.1.2.8 Outstanding Message

Outstanding message means that a message is in the midst of a message transfer (for simple mode) or waiting for a corresponding acknowledgement (for reliable mode), i.e. "outstanding" starts when the valid message number is given to the message and ends when the message number is retrieved. If the number of outstanding messages exceeds the outstanding message limit for each Software element, the messaging system shall return the error code `EBUSY`. A messaging system implementation may send multiple messages in parallel as long as it does not exceed the outstanding message limit for the same source SEID. Outstanding message limit is implementation dependent and its maximum value is 256.

3.2.2 Transport Adaptation Module (TAM)

3.2.2.1 Service Description

The part of Messaging System which is medium dependent is called the TAM. The TAM manages message fragmentation, and the message ordering and error recovery process if needed. For these purposes it defines a packet format and a protocol.

The TAM sends and receives data on the IEEE 1394 bus in the range of the FCP command register. The TAM is only notified of indications within this range which have the HAVi CTS code. A Messaging System which has enrolled for indications in the FCP address range does not influence or block other software elements from using the CMM to enroll for indications in that address range.

3.2.2.2 Fragmentation

Fragmentation is performed according to the network capabilities. For IEC 61883/1394 this service does not exist. Therefore a fragmentation service is used in the TAM to perform fragmentation on the messages generated by the Messaging System.

The TAM packet header has a part dedicated to fragmentation. If fragmentation is used then the TAM has to serialize message transmission for a particular destination node (i.e., all TAM packets for the current message will be sent before processing the next pending message for the same destination node). The TAM can parallelize message transmissions for different destination nodes.

3.2.2.3 Message Ordering

Message ordering is performed according to the network technology. For IEC 61883/1394 this service does not exist. Therefore an ordering service is performed by the TAM.

A continuity counter present within the TAM packet header is incremented (modulo 64) before each TAM packet transmission. On the receiving side, the continuity counter is processed in relation to the source GUID extracted from the 1394 packet header (and to point to the sender context).

3.2.2.4 Mapping of TAM onto the 1394 Transaction Layer

TAM packet write requests are mapped into 1394 transaction write requests (possibly via the CMM). When writing several TAM packets to a single destination a TAM shall not parallelize transactions: before generating a transaction write request to a destination it has to wait for completion of the previous transaction (to that destination).

3.2.2.4.1 IEC 61883 FCP Packet

TAM data packets are sent or received according to the IEC 61883 FCP protocol and format (see [4]).

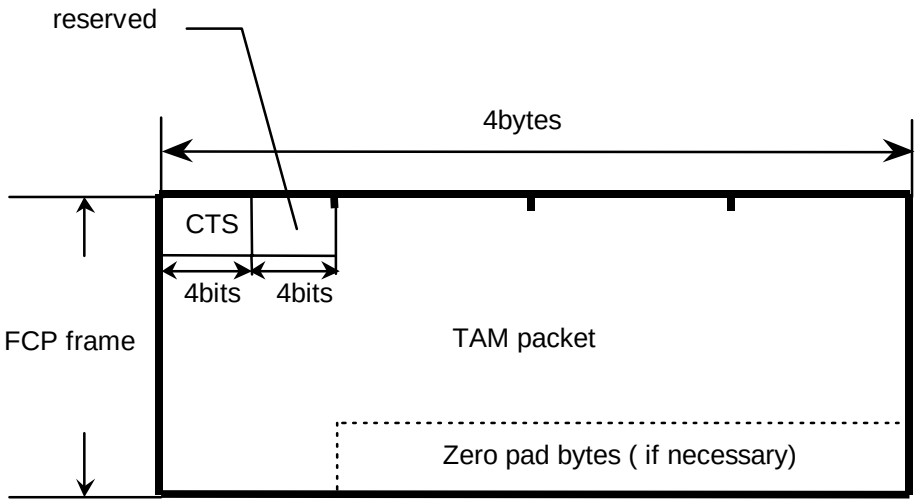

Figure 11. IEC 61883 FCP Packet Structure

A new CTS code is reserved for HAVi message transport. The following four bits after the CTS code are also reserved by HAVi; they shall be set to 0. Note that Figure 11 shows a generic IEC

61883 FCP packet that is padded with zero bytes to make the packet size a multiple of 4 bytes. In compliance with 1394-1995 the TAM layer shall indicate in the length field of the 1394 header only its own payload. The padding bytes, if any, are not included in this length.

CTS code b7	b6	b5	b4	CTS
0	0	0	0	AV/C
0	0	0	1	CAL
0	0	1	0	EHS
0	0	1	1	HAVi
		⟨		⟨
1	1	0	1	(Reserved)
1	1	1	0	Vendor Unique
1	1	1	1	Extended CTS

Figure 12. IEC 61883 CTS Codes

Concerning the transmission of messages, only the IEC61883 FCP command register is used to store a TAM data packet.

3.2.2.4.2 TAM Data Packet Structure

In case of transmission of a HAVi message, the TAM has to provide a mechanism to manage the fragmentation of messages and to preserve message ordering. For these purposes a TAM packet contains a header which permits the receiver to assemble the message.

Syntax	Number of bits	Identifier
TAM_HaviDataPacket () {		
SequenceNum	6	uimsbf
FragType	2	uimsbf
Reserved	1	uimsbf
RetryFlag	1	uimsbf
OriginalSeqNum	6	uimsbf
Reserved	8	uimsbf
for(j=0; j<FragData size; j++){	8	uimsbf
FragData[j]		
}		
}		

Figure 13. TAM_HaviDataPacket Representation

SequenceNum is used to assemble the various fragments from a message. A TAM sender shall maintain one SequenceNum for each possible TAM receiver. The sender shall continuously increment this number after every fragment is transmitted. The number shall not be reset to zero at the end of transmission of the whole message. Therefore SequenceNum can also be used to

guarantee message ordering.

`FragType` (Fragment Type) is a 2 bit value as defined in Table 4.
`Reserved` is a one bit field that shall be set to zero.
`RetryFlag` is a one bit field that indicates if the packet is a retry or not. If set to one, the packet is a "resent" packet. If set to zero, the packet is "fresh".
`OriginalSeqNum` is a 6 bit field that indicates the original `SequenceNum` of the failed original packet. If `RetryFlag` is zero, this field shall be set to zero.
`Reserved` is a one byte field that shall be set to zero

`FragData` is a variable number of bytes and may be null but shall never exceed a maximum limit, which is dependent on the target (the device receiving the TAM packet). This maximum limit derives from the maximum 1394 data payload size the target node accepts. The "max_rec" field of the configuration ROM **Bus_Info_Block** defines this value. Thus the maximum size of the `FragData` field is computed as follows:

maximum `FragData` size = "max_rec" – TAM_header_size – FCP_header_size

where TAM_header_size = 3 and the FCP_header_size = 1.

A compliant IAV or FAV must be able to accept block write requests of at least 8 bytes; TAM data packets shall never exceed 512 bytes. For the "max_rec" **Bus_Info_Block** field, a value of 0 indicates "Not Specified", which is interpreted as being able to handle at least 512 bytes. A value of 1 indicates a capacity of 4 bytes, which is not allowed for a compliant IAV of FAV implementation. Values greater than 8 indicate a capacity of at least 512 bytes.

Table 4. TAM Fragment Type Values

FragType	Value	Meaning
SFP	0x00	Simple Fragment Packet
BOP	0x01	Begin Of message Packet
COP	0x02	Continuation Of message Packet
EOP	0x03	End of message Packet

3.2.2.5 Reliable TAM Packet Transmission

If a bus reset or a transmission error occurs during a 1394 transaction, the transaction is aborted and 1394 packets may be lost. To recover from lost packets the TAM shall implement the following mechanism:

A TAM sender increases the continuity counter of the TAM packet header (see sections 3.2.2.3 and 3.2.2.4.2). If the transaction fails (reception of a `transaction_conf` with an error code), the TAM shall repeat the packet (without increasing the continuity number). The number of consecutive retries cannot exceed three. In case of failure the Messaging System returns a `Msg::ESEND` error (see section 5.3.3). After a `Msg::ESEND` error occurs, for the next packet sent to the same receiver, the sender shall increase the TAM sequence number.

Note that a TAM sender assumes the transaction succeeded as soon as it receives a `transaction_conf` (i.e., an acknowledgement from the other side). Finally a TAM sender has to wait for a `transaction_conf` from one destination before generating another transaction to that destination (increasing the continuity number).

If the TAM sender detects a bus reset before receiving a `transaction_conf` from the 1394 transaction layer, the sender has to repeat the complete message after sending a synchronization

packet as specified in 3.2.2.6. In case of an interrupted BOP, COP or EOP packet, this requires re-sending packets beginning with the BOP packet. In case of re-sending , TAM sender shall set `RetryFlag` to one and `OriginalSeqNum` to the corresponding sequence number of the first resent SFP packet or to the sequence number of the first resent BOP, COP or EOP packet. TAM receiver shall check if the incoming packet is a "resent "or a "fresh" packet with `RetryFlag`. If the packet is a "resent" packet, also check if the packet is already received or not with `OriginalSeqNum`. If the packet is already received, the packet shall be discarded.

A TAM receiver checks the continuity number for packet reordering and re-assembly. A 1394 receiver shall not acknowledge a transaction (either in a unified[1] or split transaction[2]) before being sure the 1394 packet has successfully been received. Once the transaction has been acknowledged the receiver ensures the packet will not be discarded by any bus reset. (If the TAM uses unified transactions that are acknowledged by hardware and not by the TAM itself, it shall ensure that once acknowledged a packet is assumed to be received by the TAM and that it will not be discarded by a bus reset.) If a TAM receives twice a packet with the same continuity number, it shall discard one.

A receiver shall accept a packet with any sequence number that differs from the sequence number of the previously received packet from the same sender. If an SFP or BOP packet is received, a previous sequence of BOP and/or COP packets that has not been concluded by an EOP packet shall be discarded.

3.2.2.6 TAM Sequence Number Synchronization

The first TAM packet sent from node *A* to node *B* after *A* is powered up, is reset, or has detected a 1394 bus reset event, shall be an SFP packet with a zero payload (a "synchronization packet"). This SFP packet shall not be passed to a software element, but is used only to synchronize the TAM sequence numbers of sender (*A*) and receiver (*B*). If the synchronization packet carries sequence number *n*, then the next packet sent from *A* to *B* shall have number *n+1* (*modulo 64*). Future packets sent from *A* to *B* shall have normally increasing sequence numbers. A receiver shall accept a synchronization packet at anytime.

Consider the situation where *A* and *B* are disconnected and reconnected very quickly. *A* may be a "slow" device. Thus it never detects the disappearance of *B*. *B* may be a fast device. Thus it can detect the disappearance and the re-appearance of *A*. The sequence number *B* expects from *A* will be unknown, so *B* shall expect a synchronization packet from *A*.

3.2.3 Mapping of Function Calls into Messages

IDL is the basis for mapping software element public APIs into messages.

3.2.3.1 Mapping of an IDL Interface into the Messaging System API

The APIs of software elements (system services, FCMs and DCMs) are specified as a set of operations. When a software element wants to invoke a function of another software element, it maps the function call into a message and it sends the message to that software element.

[1] In the case of a unified transaction, the acknowledgement (that ends the transaction) is the acknowledgement of the unified write.

[2] In the case of a split transaction, the acknowledgement (that ends the transaction) is the response subaction of the remote TAM.

Afterwards, the called software element may want to send a response (if the IDL operation contains a return value, `inout` or `out` parameters). To do so it sends to the calling software element.

Some general rules are needed for mapping functions into messages. Such rules are described in the following sections.

As a starting point, consider the following IDL operation:

```
IDL_Type IDL_operation(
        param_attribute1 param_type1, param_name1,
        param_attribute2 param_type2, param_name2,
        param_attribute3 param_type3, param_name3,
        param_attribute4 param_type4, param_name4 …
```

3.2.3.2 Mapping of Function Calls into Messages

When a software element A invokes an operation (`IDL_operation`) on a software element B, the operation is mapped to a message. The Messaging System shall use the following format:

Syntax	Number of bits	Identifier
`msg_function_call () {`		
`  DestSEID`	80	uimsbf
`  SourceSEID`	80	uimsbf
`  ProtocolType`	8	uimsbf
`  MessageType`	8	uimsbf
`  MessageNumber`	8	uimsbf
`  reserved`	8	uimsbf
`  MessageLength`	32	uimsbf
`  OperationCode`	24	uimsbf
`  ControlFlags`	8	uimsbf
`  TransactionId`	32	uimsbf
`for (j=0; j<param_number;j++){`		
`  if ((param_attributej == IN) ||`		
`  (param_attributej == INOUT)) {`		
`  param_valuej`	sizej	uimsbf
`  }`		
`  }`		
`}`		

Figure 14. Function Call Mapping to a Message

`DestSEID` (10 bytes) is the identifier of the software element to which the message is to be sent. It is the software element that shall execute the function.

`SourceSEID` (10 bytes) is the identifier of the software element that generates the message. It is the software element that calls the function.

`ProtocolType` (1 byte) is the format that `MessageBody` content must adhere to. The possible values for this field are listed in Annex 11.1. HAVi defines one particular protocol, `HAVi_RMI`, based on request and response message exchanges. This protocol allows one software element to invoke an operation on another (request), and it allows the result of the operation to be returned (response).

`MessageType` as described in Table 2 (shall be `msg_reliable`).

`MessageNumber` (1 byte) is the message number. Its value is incremented according to the rules described in section 3.2.1.2.3.

`reserved` (1 byte) shall be set to zero.

`MessageLength` is the number of following bytes. This length includes the `OperationCode`, `ControlFlags` and `TransactionId` lengths and the parameter loop length.

`OperationCode` is defined for each software element API. HAVi operation codes are listed in Annex 11.6.

`ControlFlags` is an 8-bit field where only the lowest bit is currently used. This bit is called the `RequestFlag` and takes the value 0. It indicates that the message is carrying a call operation request.

`TransactionId` is provided by the requester. The receiver has to put this `TransactionId` in its response message. It allows the requester to match a response with a request in case of multiple requests to the same object.

`param_valuej` is the value of the j^{th} IN or INOUT parameter of the operation specified by `OperationCode`. The parameter size, `sizej`, includes any padding added by the conversion to CDR.

The layout of a message carrying a function call is indicated schematically below:

byte 0	byte 1	byte 2	byte 3	byte 4	...	byte 13
0011 rrrr	nnnn nnff	0Rnn nnnn	0000 0000	dddd dddd	...	dddd dddd
FCPHdr	TAMHdr		reserved	Destination SEID		

byte 14	...	byte 23	byte 24	byte 25	byte 26	byte 27
ssss ssss	...	ssss ssss	0000 0000	0000 0010	NNNN NNNN	0000 0000
Source SEID			ProtType	MsgType	MsgNo	reserved

byte 28	byte 29	byte 30	byte 31
LLLL LLLL	LLLL LLLL	LLLL LLLL	LLLL LLLL
Message Length			

byte 32	byte 33	byte 34	byte 35
oooo oooo	oooo oooo	oooo oooo	cccc ccc0
OperationCode			CtrlFlag

byte 36	byte 37	byte 38	byte 39
TTTT TTTT	TTTT TTTT	TTTT TTTT	TTTT TTTT
Transaction Id			

3.2.3.3 Mapping of Function Returns into Messages

When a software element A invokes a function on another software element B using a request

message, and if the function gathers INOUT, OUT or a return code, the called software element shall map the output of the function to a message. The Messaging System shall use the following format:

Syntax	Number of bits	Identifier
msg_function_response () {		
DestSEID	80	uimsbf
SourceSEID	80	uimsbf
ProtocolType	8	uimsbf
MessageType	8	uimsbf
MessageNumber	8	uimsbf
reserved	8	uimsbf
MessageLength	32	uimsbf
OperationCode	24	uimsbf
ControlFlags	8	uimsbf
TransactionId	32	uimsbf
ReturnApiCode	16	uismbf
ReturnErrCode	16	uimsbf
reserved	32	uimsbf
for (j=0; j<param_number;j++){		
if ((param_attributej == OUT) ||		
(param_attributej == INOUT)) {		
param_valuej	sizej	uimsbf
}		
}		
}		

Figure 15. Function Return Mapping to a Message

`DestSEID` (10 bytes) is the identifier of the software element to which the message is to be sent. It is the software element that calls the function and that will get the function output.

`SourceSEID` (10 bytes) is the identifier of the software element that generates the message. It is the software element that has executed the function that sends the output.

`ProtocolType` (1 byte) is the format that `MessageBody` content must adhere to. The possible values for this field are listed in Annex 11.1. HAVi defines one particular protocol, `HAVi_RMI`, based on request and response message exchanges. This protocol allows one software element to invoke an operation on another (request), and it allows the result of the operation to be returned (response).

`MessageType` as described in Table 2.

`MessageNumber` (1 byte) is the message number. Its value is incremented according to the rules described in section 3.2.1.2.3.

`reserved` (1 byte) shall be set to zero.

`MessageLength` is the number of following bytes.

`OperationCode` is defined for each software element API. HAVi operation codes are listed in Annex 11.6.

`ControlFlags` is an 8-bit field where only the lowest bit is currently used. This bit is called the `ResponseFlag` and takes the value 1. It indicates that the message is carrying an operation response.

`TransactionId` is provided by the requester. The receiver inserts this `TransactionId` in its response message. It allows the requester to match a response with a request in case of multiple requests to the same object.

`OperationReturnCode` contains the return code of the IDL operation.

`ReturnApiCode` and `ReturnErrCode` together contain the return code (i.e., `Status`, see 5.1.2) of the IDL operation.

`reserved` (4 bytes) shall be set to zero.

`param_valuej` is the value of the j^{th} OUT or INOUT parameter of the operation specified by `OperationCode`. The parameter size, `sizej`, includes any padding added by the conversion to CDR.

The layout of a message carrying a function return is indicated schematically below:

byte 0	byte 1	byte 2	byte 3	byte 4	...	byte 13
0011 rrrr	nnnn nnff	0000 0000	0Rnn nnnn	dddd dddd	...	dddd dddd
FCPHdr	TAMHdr		reserved	Destination SEID		

byte 14	...	byte 23	byte 24	byte 25	byte 26	byte 27
ssss ssss	...	ssss ssss	0000 0000	0000 00tt	NNNN NNNN	0000 0000
Source SEID			ProtType	MsgType	MsgNo	reserved

byte 28	byte 29	byte 30	byte 31
LLLL LLLL	LLLL LLLL	LLLL LLLL	LLLL LLLL
Message Length			

byte 32	byte 33	byte 34	byte 35
oooo oooo	oooo oooo	oooo oooo	cccc ccc0
OperationCode			CtrlFlag

byte 36	byte 37	byte 38	byte 39
TTTT TTTT	TTTT TTTT	TTTT TTTT	TTTT TTTT
Transaction Id			

byte 40	byte 41	byte 42	byte 43
AAAA AAAA	AAAA AAAA	EEEE EEEE	EEEE EEEE
ApiCode		ErrCode	

byte 44	byte 45	byte 46	byte 47
0000 0000	0000 0000	0000 0000	0000 0000
reserved			

The field “MsgType” (`MessageType`) is either 0000 0001 or 0000 0010.

3.2.3.4 Mapping of IDL Types and Parameters to Bitflows

The mapping uses a subset of CDR (Common Data Representation) from GIOP (General Inter Orb Protocol) version 1.1 [5] as the transfer syntax for the parameters. Each parameter is byte aligned according to its natural length, e.g., a parameter of length four bytes should start on a four byte boundary. The following restrictions are used:

- The byte ordering is BIG ENDIAN: MSB (Most Significant Byte) first.
- Since the types of all parameters in HAVi messages can be unambiguously interpreted, CDR type codes are not used.
- Concerning the complex types, repository ID, name and member name are not supported. For example the `tk_struct` is followed by the list of member types of the structure.
- The fixed size for the `wchar` IDL type is 2 bytes according to UNICODE UTF-16 and ISO 10646 UCS-2.
- Padding bytes shall be set to zero.

The starting point of the CDR mapping is the field after `MessageLength` as described in the section on “General Message Format” (see 3.2.1.2.4, Figure 10). Specifically, the area where the CDR mapping is applied is the `MessageBody`.

3.2.3.5 Synchronous Message Transfer Mode

The message passing API will provide a synchronous service allowing a caller to block until a response is received. As shown in the following figure, the caller asks to send a function call through the message passing API. The local Messaging System sends a request message (using reliable mode) and waits for the response or a timeout condition. The remote Messaging System receives the request and passes it to the destination software element. The destination element sends its function response message using a normal send in simple or reliable form. The requester’s Messaging System receives the response and transmits it to the requester.

Note that the response message is delivered to the caller software element by completing the `msg_send_sync` invocation. If the mode of the response message is reliable, the requester’s Messaging System sends a `msg_reliable_ack` (success to transmit) or `msg_reliable_noack` (failure to transmit) related to the reliable response message, to the remote Messaging System, after the requester’s Messaging System receives the response message.

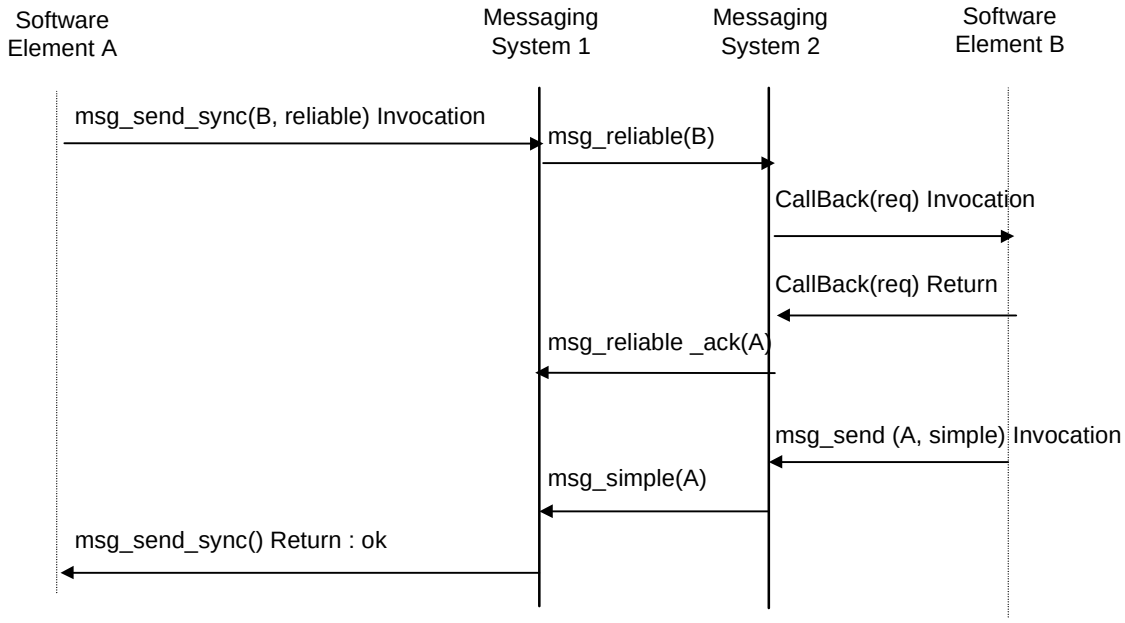

Figure 16. Example of Synchronous Message Transfer

Figure 16 shows an example where the response message is sent in simple mode. It is also possible that the destination element sends the response message in reliable mode.

3.2.4 Implementation Guidelines and Suggestions

3.2.4.1 GUID to phy_id Mapping

The Messaging System uses the GUID (either directly or indirectly through the SEID) to identify a network device in a stable way. It is the responsibility of the Messaging System (and/or the CMM1394) to build and maintain the association table between the GUID and the 1394 phy_id. The building of such a table may be very slow (due to the need to read the GUIDs of all nodes after the detection of the bus reset). To increase the efficiency of this discovery process, the following algorithm, which allows the tracing of the 1394 phy_ids based on the self_id packets, may be used.

After a bus reset the physical addresses of 1394 nodes may change. However each node does not receive information about the addresses other nodes have obtained. HAVi uses a persistent method of node identification based on the 64-bit node unique identifier (GUID). One method of determining the new physical address of a node, given its GUID, is to read the GUID on each node of the network until a match is found. The disadvantage of this method is that it may generate a lot of traffic (in particular if all the nodes use the same technique) and so may be slow.

The aim of the algorithm described here is to re-identify the nodes after a bus reset by using *self_id* packets, and so find the new physical addresses of all nodes. The GUID can then be *directly* read on the node to verify the result.

The main principle is to construct the connection tree of the network before and after the reset, and with this data, to construct a translation table indicating for each node its new number or whether the node has disappeared or is new.

There are two stages:

- construction of a connection tree, and
- construction of a translation table.

3.2.4.1.1 Connection Tree Construction

The aim of this stage of the algorithm is to build the connection tree of nodes of a 1394 bus with the *self_id* packets.

Note: The 1394 standard assumes a similar procedure to construct the speed map with the topology map.

Information available initially is:

- the self-renumeration strategy on a 1394 bus,
- for each port of each node, whether there is a node connected to the port and whether it is a child or the parent.

The following paragraph is just a simplified summary of the 1394 self-re-numeration strategy, for more information see section 3.7.3.1 of *IEEE Std 1394-1995 High Performance Serial Bus, 1996-08-30*.

Schematically, after the root election, the node which is root wants to determine its number. When a node wants to know its number, it asks the number of each child going from the lower port to the upper port. The node's number is then the last self_id packet viewed on the bus plus one. This number is then sent via a self_id packet on the bus. For example:

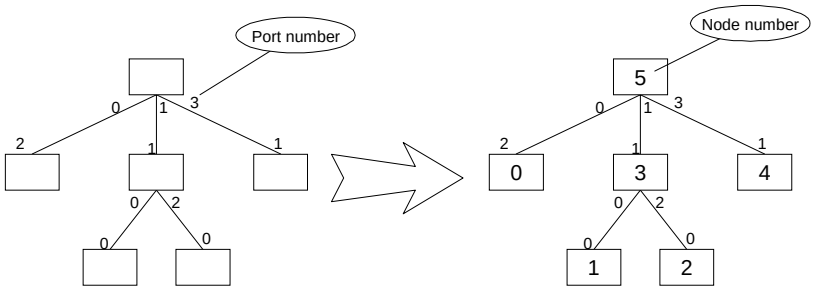

Figure 17. Self-renumeration Strategy

To build a connection tree, perform the following steps using the self_id packets generated during 1394 self-renumeration.

1. Build two node sets: one called *parent-set* containing nodes which have at least one child and one called *child-set* containing nodes which have no child.
2. Take the smallest node in the *parent-set* (let the number be *P*).
3. The number of children of *P* is known, their node numbers are the greatest nodes in the *child-set* with a number less than *P*. Their port is also known (the lowest is on the lowest child port...). Remove the children of *P* from the *child-set*.
4. Move *P* from the *parent-set* to the *child-set*.
5. While the *parent-set* is not empty, go to step 2.

3.2.4.1.2 Example

Using the network in Figure 17, the known information is:

Node	Parent Port	Child Ports	Unconnected Ports
0	2		0,1
1	0		1
2	0		
3	1	0,2	
4	1		0
5		0,1,3	2

Algorithm actions and results:

Description	Parent-set	Child-set	Result
Beginning	3,5	0,1,2,4	
The smallest parent is 3. It has 2 children. The greatest children with number less than 3 are 2 and 1.	5	0,3,4	The node 1 is connected to the port 0 of the node 3. The node 2 is connected to the port 2 of the node 3.
The smallest parent is 5. It has 3 children. The greatest children with a number less than 5 are 4, 3 and 0.			The node 0 is connected to the port 0 of the node 5. The node 3 is connected to the port 1 of the node 5. The node 4 is connected to the port 3 of the node 5.
The parent-set is empty, so the tree is built.			

As this example shows, with just the information contained in the self_id packets one can build the connection tree.

3.2.4.1.3 Translation Table Construction

The aim of this stage of the algorithm is to build the translation table of the node numbers which give the new node numbers after the reset.

Information available initially is:

- trees built with the above algorithm before and after the reset but with the current node (which realizes this process) as root,
- the old number of the current node and its new number,
- normally a node cannot be swapped with another without producing at least 2 resets: so nodes can only appear or disappear in a network, not move.

We assume that a node will be a child of another node if it is connected to it and it is more distant than its parent from the root.

The main principle of the algorithm is that on each port:

- before a reset, there is the node number *A*, and after the reset there is the node number *B*, so the new number of *A* is *B*,
- before a reset, there is the node number *A*, and after the reset there is no node, the node number *A*, and all the children of this node, have been removed from the network,
- before a reset, there is no node, and after the reset there is the node number *A*, the node number *A*, and all the children of this node, have been added to the network.

The following steps are performed for all ports of all nodes. It begins with the local node which serves as a common point in the before and after connection trees.

The basic function is *ProcessNode (old, new)* which processes the node numbered *old* before the reset and *new* after.

1. if *old* differs from *NONUMBER*, go to step 6
2. take the first port
3. if there is a child numbered *C* on this port, add *new* to *added-node-set* and call *ProcessNode(NONUMBER, C)*
4. if there is another port, go to step 3
5. go to step 13
6. put the relation between *old* and *new* in the *translation-table*.
7. take the first port
8. if there is no child on this port in the old and in the new tree, go to step 12
9. if there is no child in the new tree, add the old child number (with all its sons) to *removed-node-set* and go to step 12
10. if there is no child in the old tree, add the new child number (*C*) to *added-node-set* and call *ProcessNode(NONUMBER, C)*
11. else call *ProcessNode(*the old child number, the new child number*)*
12. if there is a next port, take it and go to step 8
13. end of this node process (all its sons have been processed).

Note: The NONUMBER is used to indicate to the function that the node given in new was not present before the reset and so has no old number, it allows processing of its children which are automatically new too.

An improvement for better security can be added before point 6: to help assure that *old* and *new* are the same node, one can verify that their features are the same (number of ports and speed capability).

3.2.4.2 Message Size Guidelines

Since all TAM packets for the current message will be sent before processing the next pending message for the same destination node (see section 3.2.2) very large messages should be used with caution. A very large message will block important global events and/or messages of other software elements. It may cause a “freezing period” that disrupts the functioning of software elements on the same node – even software elements that have no relation to the sender of the very large message.

For example, even in the best case of a message sent by 1394 asynchronous writes of 512 byte packets on a 100 Mbps network, the full length message (0xffff ffff bytes) will take 343.6 seconds (5.7 min) to transfer. This time is likely too long to wait for the next message, moreover it breaks the timeout rule for the Messaging System (30 sec).

In the worst case, when 63 nodes are simultaneously sending full length messages on the 100 Mbps network with 80% isochronous traffic, the transfer will take 36.7 hours. In this worst case, the largest possible message size which does not break the timeout is approximately 960 Kbytes.

(*Note* – these examples are theoretical and the practical performance may be lower.)

For safety, HAVi makes the following recommendations regarding HAVi message sizes:

- to not break the Messaging System timeout, it is recommended to use messages of size less than 512 Kbytes.

- to ensure tolerable response for users, it is recommended to use messages of size less than 64 Kbytes. (This takes about 2 seconds in the worst case.)

Messages larger than these sizes should be used with caution. The bulk transfer APIs (see section 5.14) are recommended if message size exceeds 64 Kbytes.

Messages larger than 64 Kbytes can be refused or aborted by the Messaging System to avoid a timeout. The size which is refused or aborted is Messaging System implementation dependent. It is recommended that the Messaging System makes a best effort to send very large messages.

3.2.4.3 Software Element Design

The basis of a HAVi network is that requests are sent from Software Element to Software Element, actions are taken, and corresponding responses are returned. How the Software Element handles these requests, actions and responses is largely up to the designer; however, it is critical that the designer consider the Software Element's use of HAVi messaging carefully.

Received message requests often trigger various actions. If the actions are implemented in a way which blocks the thread that receives new messages, no new message will be processed until the previous action completes. If an action takes a long time to complete, such a design may result in poor performance. Even worse, if a blocking action includes a request to another software element, a deadlock condition may occur. An example of deadlock is when two Registries simultaneously query each other, but cannot receive the queries, since they cannot complete their previous actions until the queries are answered.

In general there are two solutions to the problem: one is to use one or more additional threads to send synchronous requests. Another is to send requests asynchronously. A small number of multiple threads may be acceptable for simple situations, but multiple threads may be costly. Also, in complex situations, it may be difficult to predict the maximum number of threads needed. By implementing asynchronous requests a Software Element can handle complex situations efficiently.

When making an asynchronous request, a Software Element must store the transaction ID and possible other information about the original request in some sort of table. Normally the entries are removed every time a matching response is received. In the case that a matching response is not received, the corresponding request information will not be automatically removed from the table, causing a potential memory leak. Worse yet, if the SE never receives the response, it may not be able to complete its action, causing parts of the HAVi system to freeze.

To avoid these problems it is recommended that Software Elements that implement asynchronous requests also implement some sort of a timeout mechanism. The timeout mechanism would ensure that actions are completed, and that request data does not build up in tables. (Note that for synchronous requests, such a timeout mechanism is provided by the Messaging System.)

There are many reasons that may cause responses not to be received. One reason may be that the destination Software Element, or destination device, is removed during a request. In these cases a timeout will occur, completing the request. Unfortunately, the timeout may not occur until after a long delay, resulting in very long response times.

In order to respond more quickly to removed devices or software elements, the Software Element designer may choose to activate a watch (`msgWatchOn`) on the target before sending a request message. Another option would be to detect the disappearance of the target by registering for events such as `NetworkReset`, `GoneDevices`, and `GoneSoftwareElement`.

3.2.4.4 Unknown source GUID / node ID (informative)

A HAVi device may receive a TAM packet/message with an unknown source GUID and unknown source node_id (the device has not been detected within the bus or the IEEE1394 bus_ID indicates that the packet originates from a remote bus).

To allow answering to such a message, the following option may be implemented:

The incoming message will be processed and dispatched by the message system. If the incoming message is a request, then the GUID-node_ID association is temporarily kept by the target device in order to be able to send back a response. Once the HAVi response message is sent, the temporary GUID-node_id association is removed. Note that this temporary GUID/node_id entry is private to the messaging system and not known to the CMM. For this reason it is not reflected in the GUID list and no device changed events are posted.

3.3 Event Manager

The Event Manager provides an event delivery service. An event is a change of state of a software element or of the home network. For example, adding or removing a device implies a change of state of the network and is likely (but not necessarily) to trigger an appropriate event. The delivery of an event is done either locally within a single device or globally to all devices in the network; local or global delivery is selectable by the event poster. To support this service, the Event Manager functions as an agent to help assure the event posted by a software element will reach all software elements that care about the event. If a software element wishes to be notified when a particular event is posted, it must register such intention with its local Event Manager. Each Event Manager maintains an internal table containing the list of events registered by software elements. When a software element posts an event, it does so via a service provided by Event Manager. The Event Manager checks its internal table and notifies those software elements that have registered this event. Software elements that do not register the event will not receive a notification. If the event is posted globally, the local Event Manager also relays the event to all remote Event Managers in the network. Each remote Event Manager performs the same lookup and notifies the registered local software elements. An Event Manager notifies software elements by using the HAVi Messaging System; in particular, it sends a notification message to the software element that is to be notified.

An event has an optional buffer that can be used to pass information related to the event. For example, consider an input device with multiple buttons, a button pressed event could be generated when the user presses a button. The software element that is notified of this event may also be interested to know which button was pressed. The event poster can optionally put additional

information in the buffer and let the Event Manager pass this information to other software elements. A software element would get the optional information as part of the notification process and it is the software element's task to interpret the information.

3.3.1 Mapping IDL Events to the Event Manager API

The specifications of the HAVi software elements contain IDL definitions of events they generate. Events are described by an IDL procedure with only input parameters and a `void` return value. In general they have the following format:

```
void EventName( in param_type1 param_name1,…);
```

When a software element posts an event it has to map the IDL event definition to parameters of the `EventManager::PostEvent` API. The rules for mapping IDL event definition are as follows:

- The IDL procedure name `EventName` should be mapped to the `EventId` *event* parameter of the `PostEvent` API.
- The `global` parameter must indicate the distribution of the event as given in Annex 11.9.
- All event parameters must be provided in the `eventInfo` parameter. They should be encoded using the Common Data Representation (CDR) standard in the same order they are given in the event definition. The first byte of `eventInfo` is considered the "zero index" for natural boundary alignment.

3.4 Registry

The Registry is a system service whose purpose is to manage a directory of software elements available within the home network. It provides an API to register and search for software elements. The Registry service shall be present on each IAV and FAV. Within one device any local software elements can describe itself through the Registry. If a software element wants to be contacted, it must register with the Registry. System software elements shall be registered so that they can be found and contacted by any software element in the network.

The Registry maintains, for each registered object, its identifier (SEID) and its attributes. The Registry also provides a query interface which software elements can use to search for a target software element according to a set of criteria.

3.4.1 Registry Database

Each Registry contains tables describing local software elements (software elements within the same device). The logical database is viewed as the set of all these tables. Each Registry service offers the possibility to query this database.

Each Registry database has the structure indicated in Table 5:

Table 5. Registry Database Structure

Syntax	Number of bits	Identifier

`Database() {`
`for (e=0; e<N;e++){`
`  SEID`
`for (a=0; a< M; a++){`
	`80`	`uimsbf`

attribute() } } }		

Element	Description
SEID	The *software element identifier* is an 80 bit number representing the unique identifier of a software element within the home network. It is provided to the software element by the Messaging System API . Any software element can send a message to another using this software element identifier.
Attribute	The *software element attributes* that characterize the software element. All attributes can be gathered in a table and have the structure indicated in Table 6.
N	The max number of entries (registered software elements) in the database.
M	The max number of attributes for a software element

3.4.2 Registry Attributes

A Registry attribute has the following structure:

Table 6. Registry Attribute Structure

Syntax	Number of bits	Identifier

Attribute() {
  Class
  Name
  Size
for (j=0; j<Size;j++){
  value[j]
  }
}
	
	1
	31
	32
	
	8
		
		uimsbf
		uimsbf
		
		uimsbf

Element	Description
Class	The *attribute class* is private (1) or system (0). In case of a system attribute the name values (see below) are well defined. In case of private attributes, the name must be associated with some other attribute like Device Manufacturer or Software Element Manufacturer. This means that a query on a private attribute has to contain some other criteria to avoid private name conflicts.
Name	The *attribute name* indicates the name of an attribute. It is represented by a number. The next section presents the system attribute list. An IDL type is associated with each *system* attribute (see Table 7 below), this type indicates allowable values for the attribute.
Size	The *attribute size* gives the number of bytes of an attribute value.

valuej	*valuej* indicates the j[th] byte of the attribute value. The attribute value is formatted as specified in the CDR transfer syntax ([5] – chapter 12.3) with the following restrictions: (1) The byte ordering is BIG ENDIAN: MSB (Most Significant Byte) first. (2) Since the types of the arguments are assumed to be known by the caller, the attributes can be unambiguously interpreted and CDR type codes are not used. (3) The fixed size for the wchar IDL type is 2 bytes according to UNICODE UTF-16 and ISO 10646 UCS-2. (4) The first byte of the attribute value (*value0*) is considered the "zero index" for natural boundary alignment.

The following table identifies the predefined system attributes.

Table 7. Predefined Registry Attributes

Attribute Name	IDL Type	Fixed or Variable Size	Max size (bytes)	Presence	SV or MV
ATT_SE_TYPE	SoftwareElementType	F	4	M	SV
ATT_VENDOR_ID	VendorId	F	3	M[A]	SV
ATT_HUID	HUID	F	28	M[A]	SV
ATT_TARGET_ID	TargetId	F	18	M[A]	SV
ATT_INTERFACE_ID	InterfaceId	F	2	M[A]	SV
ATT_DEVICE_CLASS	DeviceClass	F	4	M*	SV
ATT_GUI_REQ	GuiReq	F	4	O	SV
ATT_MEDIA_FORMAT_ID	MediaFormatId	F	8	O	MV
ATT_DEVICE_MANUF	DeviceManufacturer [3]	V	106	M*	SV
ATT_DEVICE_MODEL	DeviceModel	V	106	O	SV
ATT_SE_MANUF	SoftwareElementManufacturer	V	106	O	SV
ATT_SE_VERS	SoftwareElementVersion	F	4	M[S]	SV
ATT_AV_LANG	AvLanguage	F	4	O	SV
ATT_USER_PREF_NAME	UserPreferredName	V	38	M*	SV

symbol	*meaning*	*symbol*	*meaning*
M	Mandatory	F	Fixed size
M*	Mandatory for DCM and FCM	V	Variable size (up to the maximum size)
M[A]	M* + Application Module	M[S]	M* + system components
O	Optional	SV/MV	Single valued/Multi-valued

The set of attributes associated with a software element in the Registry database may contain at most one occurrence of those attributes that are "single valued" (indicated by SV in the above table).

[3] The size of a wide char (wchar in IDL) will be two bytes (see "Messaging System "chapter).

3.5 Device Control

In a HAVi network, a DCM (Device Control Module) should exist for each device known in that network. The DCM provides an interface to the device by presenting it as a software element in the HAVi architecture. Associated with a DCM are zero or more FCMs (Functional Component Modules). FCMs are software elements that represent the different functional components contained within a device. The number of FCMs within a DCM is flexible and may vary over time. A DCM can be asked for the list of FCMs it currently contains.

A DCM may also use an FCM to represent the functionality of an *external legacy device*. (Such a device is one connected via an external plug to the device represented by the DCM itself.) How the DCM identifies the external device is proprietary to the DCM. It is recommended that DCMs do not accept connection requests on external plugs that are connected to external legacy devices "replaced" by FCMs.

Applications can query the Registry to find the devices and functional components available, and to obtain their software element identifiers. This allows the application to interact with the device via the DCM and the FCMs. A DCM and its FCMs are obtained from a DCM code unit for the device. DCM code units are installed by FAVs and IAVs. Installation of a code unit results in the installation of the DCM and all the associated FCMs. DCM code units can be written in Java bytecode, in which case they can be installed on any FAV device, or in some native code, in which case they can be installed only on (and by) some FAV or IAV that can execute that code. More concretely:

- *DCM code unit*. A piece of code related to a HAVi device. DCM code units are handled and installed by FAV and IAV devices. When a DCM code unit is installed, the DCM code unit will in turn install the DCM for the device; the DCM in its turn will install the FCMs for the functional components currently available within the device. DCM code units can be written in Java bytecode or a native code. DCM code units may come from different sources (e.g., embedded in an IAV, stored in a BAV, or from the Internet). The format of DCM code units is described in section 7.4.1.

- *Device Control Module (DCM).* A software abstraction of a device providing device specific functionality to the HAVi software environment and applications. HAVi applications will not communicate with a device directly but through the DCM of the device (or one of the FCMs). A DCM is an HAVi object in the sense that it is registered in the Registry and it can communicate with other HAVi objects via the HAVi Messaging System.

- *Functional Component Module (FCM).* A software abstraction of a functional component providing the functionality of that functional component to the HAVi software environment and applications. HAVi applications will not communicate with a functional component directly but only through the FCM (this is at least the model used to present the relation, the FCM implementation may communicate with the CMM directly). An FCM is an HAVi object in the sense that it is registered in the Registry and it can communicate with other HAVi objects via the HAVi Messaging System.

For the different types of HAVi devices, DCMs play a different role.

- An IAV device may host one DCM representing itself and may host one or more DCMs representing LAVs (or BAVs operating in *LAV mode*, see below).

- An FAV device shall host one DCM representing itself and may host one or more DCMs representing LAV devices and/or BAV devices.

- An LAV device does not have any notion of DCMs. When attached to a HAVi network where one or more FAV or IAV devices know how to handle the LAV, one of them has to provide the DCM code unit to make the LAV available to other HAVi components. How this is done and how the DCM/FCMs communicate with the LAV device is completely proprietary to the manufacturer of the FAV or IAV device.

- A BAV device does not host any DCMs, but provides a DCM code unit in Java bytecode. When attached to a HAVi network with one or more FAVs, one of them uploads and installs the DCM code unit to make the BAV device available to other HAVi components. Installation of the DCM code unit results in the installation of the DCM and all FCMs related to the device. In this situation, the BAV is said to operate in BAV mode. The IEEE 1394 communication between the DCM/FCMs and the BAV goes via the standardized CMM1394 API. However the content of the IEEE 1394 messages interchanged by the DCM/FCMs and the BAV is proprietary to the BAV manufacturer.

- When attached to a HAVi network with no FAV devices but with an IAV device that knows how to handle that BAV, an IAV device can provide a DCM code unit itself to make the BAV device available to other HAVi components. The BAV is then said to operate in *LAV mode* in which case the situation is the same as for an LAV device.

Besides the APIs to control the device (and its functional components), a DCM may also contain a device specific application. Through this application, a device manufacturer can provide the user a way to control any special feature of the device in a way decided by manufacturer, without the need for standardizing all these features in HAVi. It is provided via the API of the DCM and may be provided at two different levels. For Level 1 interoperability, the DCM may provide an API for the Data Driven Interaction mechanism. For Level 2 interoperability, the DCM provides an API for FAV devices to upload a havlet code unit. Such a code unit consists of bytecode that can be installed by an FAV device and results in a havlet. The format of havlet code units is described in section 7.4.3.

A havlet obtained from a DCM is a Level 2 (i.e., Java bytecode) application that provides a user interface for control of the device associated with the DCM. For the actual control of the device, the havlet communicates with the DCM (using its standard interface, and possibly, proprietary extensions). A havlet is a HAVi object in the sense that it can communicate with other HAVi objects via the HAVi Messaging System.

A havlet code unit is always uploaded and installed on the initiative of an FAV device. An example scenario is shown in Figure 18.

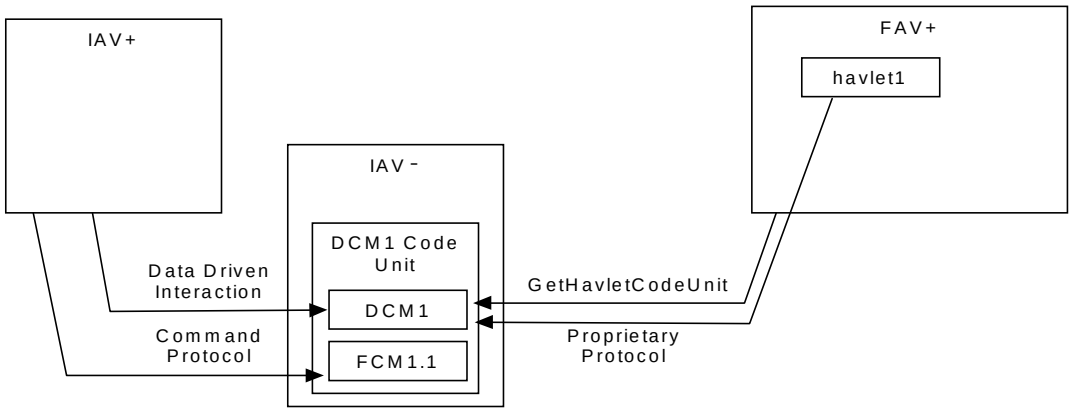

Figure 18. Havlet Upload

The figure above shows an IAV- (without a display) that provides a DCM for some device. An IAV+, with a display, can control that device in the following way. An embedded application on IAV+ interacts with the Functional Component Module *FCM1.1* (or Device Control Module *DCM1*) through HAVi messaging. For device specific functionality, IAV+ can allow user interaction with *DCM1* through the Data Driven Interaction mechanism. An FAV+ could handle the IAV- in the same way. But as shown in the picture, an FAV+ also has the ability to upload a havlet. Therefore it interacts with *DCM1* to get the havlet code unit. After installation of this bytecode, *havlet1* is available. To control the device, the havlet communicates with *DCM1* via HAVi messages, however the content of these messages may be proprietary to the manufacturer of the DCM and havlet.

DCM Managers are responsible for installing DCM code units for new devices attached to the HAVi network. A device may consist of more than one functional component; e.g., one device may consist of a tuner and a VCR. For a BAV or LAV device, the installation of its DCM code unit always takes place on a per device basis, not for each DCM component separately, so, for a DCM manager a DCM code unit is a single entity for installation, removal and replacement of DCM components. Furthermore, one DCM code unit corresponds to a single BAV or LAV device, i.e. it contains all DCM components for that device.

For BAV devices, DCM code units play a special role. A BAV device provides a single DCM code unit as a piece of bytecode (as part of the SDD) and a standardized mechanism for communication with an FAV. An FAV device can upload and install a DCM code unit (by the DCM Manager). Installation of a DCM code unit results in the instantiation of all DCM components representing that BAV device.

The communication relation between the components of a BAV DCM code unit and the system components of its hosting controller are depicted in the figure below. The CMM provides a basic service for elementary communication with the BAV device. The DCM Manager detects the attachment of a new device and obtains the GUID of that device; it can then communicate with the device to obtain basic information about the device and to retrieve the DCM code unit. The DCM Manager installs (and later removes) the DCM code unit. This installation results in the DCM and FCM objects that communicate with the device via the CMM. On installation, DCMs and FCMs make themselves known via the Registry so that they can be used by other applications via the HAVi Messaging System.

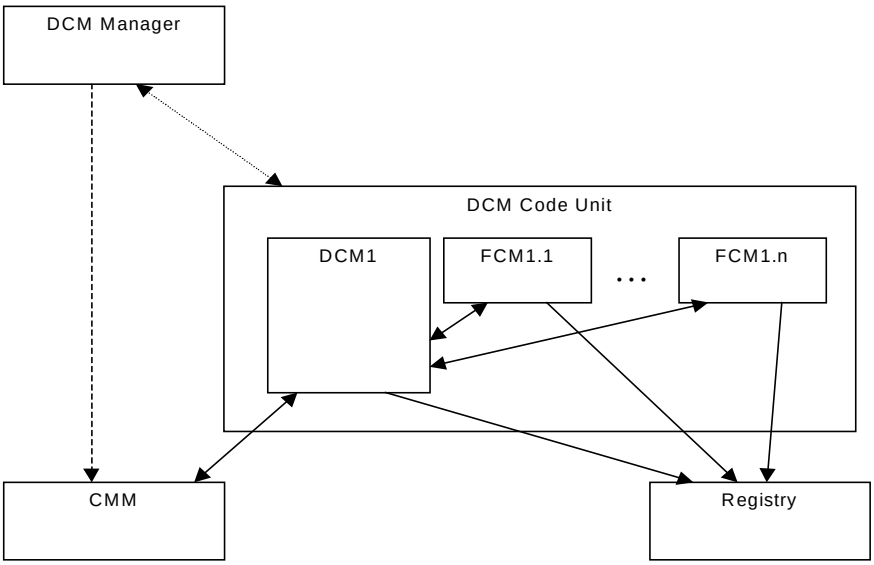

Figure 19. DCM Installation

The communication between the DCM components and the CMM is based on the GUID of the BAV/LAV device. The CMM requires only the GUID for communication with the device driven by DCM components. The CMM does not need to have any knowledge about the structure of functional components within a BAV device. This simplifies the communication between an FAV and a BAV device, makes the standardization effort smaller, and increases the possibilities for using proprietary protocols. The DCM components themselves are responsible for the proper use of the CMM, i.e. for interleaving the communication in a proper way and the distribution of messages received from the BAV device to the proper DCM components.

The contents of BAV DCM code units must be standardized so that each DCM Manager can handle DCM code units from arbitrary BAV devices. The DCM code unit is a kind of self-extracting package in Java bytecode; it provides the DCM Manager with handles for installation and removal. The DCM Manager just calls the install handle and provides the device's GUID (for future communication between the device and the DCM via the CMM). The DCM code unit itself is responsible for installing all its DCM components. Similarly, the remove handle provides the DCM Manager with a handle to remove all DCM components within the DCM code unit. This allows maximum freedom for BAV manufacturers in structuring their DCM code units.

3.5.1 Device Control Modules

A DCM, is a software abstraction of a device providing device specific functionality to the HAVi software environment and applications. HAVi applications will not communicate with a device directly but through the DCM of the device or one of its FCMs. A DCM is a HAVi object in the sense that it is registered in the Registry and can communicate with other HAVi objects via the HAVi Messaging System. DCMs and FCMs are registered with their type and their HAVi unique identifier (HUID). The HUID allows applications to find the DCM or FCM after partial system unavailability (as when the device represented by the DCM is momentarily removed from the network).

3.5.1.1 General

A DCM provides a set of basic methods for device control and observation. This API can be used by HAVi system components as well as any application. It includes the following functionalities:

- Device representation – Provides a (Level 1) visual representation of the device that can be displayed to the user.
- HUID information – Provides the HUID of the DCM as well as those of the corresponding FCMs.
- User Preferred Name – Allows assigning and retrieving of a user specified device name.
- Power management – Provides means to turn the power of the device on or off.
- Native Commands – Provides means to control the device in its native command set (CAL, AV/C, etc).

3.5.1.2 HAVi Unique Identification

In HAVi, it is possible to write applications and system components (like a Resource Manager) that always use the same functional component or that look for the same functional components used in a previous session. To support this capability, applications must have a unique and persistent way to identify the DCM representing a device and to identify the FCMs representing functional components on that device. This identification is called the HAVi Unique ID, or HUID. Ideally the HUID would be persistent across bus resets and network reconfigurations, however this is not guaranteed under all circumstances and a DCM for a certain device may be assigned different identifiers at different times.

For example, it is possible to write an application that sets up a user preferred configuration for watching a video: using the small TV set, the upper VCR deck of the 3 deck VCR box and the special wireless headphone set, all situated in the living room. This configuration would be indicated by the user once, and each time the application runs, the specified configuration must be set up.

Note that identification of DCMs and FCMs is subtly different from identification of devices and functional components. The identification of a DCM/FCM indicates the device as well as the functionality of that device provided by the DCM/FCM. This difference can be seen by the following example:

Assume a network with a BAV CD-ROM player, an FAV and an IAV device. The DCM code unit of the BAV device is uploaded and installed on the FAV device and provides the full functionality of a CD-ROM player. The DCM code unit gets a HUID based on its 1394 GUID. An application running on the IAV device stores this HUID and when it wants to use the CD-ROM player it can retrieve the DCM based on this HUID (when it is available in the network). Now, the FAV device is removed. The IAV is able to control the CD-ROM player, but has only a DCM that provides the CD-Player functionality of the CD-ROM. This DCM should not get the same HUID as the BAV's DCM since the functionality provided by the DCM is different (although it represents the same device). This prevents an application from mistaking this DCM for the DCM of the CD-ROM player. However, instead of getting a CD-ROM, the application now only gets a CD-Player. (In this situation the difference can be easily detected by the type of the DCM, however, the problem may be more subtle in general.)

For FAVs, IAVs and BAVs (in BAV mode) and their functional components, the HAVi Unique Identifiers allows a unique and persistent identification, which means:

Unique – By the HUID, an application is able to detect:

- Whether a device or functional component is the same as a device or functional component it has used before: if a software element with HUID *H* represents the device or functional component *D*, then it will *always* be the case that any software element with HUID *H* represents *D*.

  This means that the HUID must be able to identify the device (the black triple deck VCR in the family room, instead of the VCR in the kitchen); as well as the functional component on that device (the upper VCR deck instead of the middle or lower one).

- Whether a DCM or FCM is the same as a DCM or FCM it has used before: if a software element with HUID *H* has some API *A*, then it will *always* be the case that any software element with HUID *H* has API *A*.

  This means that it also should be able to identify the specific API corresponding to the device or functional component. In general, a device can be represented by different DCMs and FCMs (at different points in time) which may have different APIs; e.g., different proprietary extensions of the standard command sets.

Note that the type (of device or functional component), Vendor ID (of the device) or GUID or even a combination of these is not sufficient to uniquely identify an FCM. There may be several identical (of the same type and vendor) devices in the network containing several identical functional components (e.g., a double deck cassette player, or a triple deck VCR).

Persistent – The HUID is persistently unique over network resets, device removal, re-attachment as well as complete network shutdowns: if a device or functional component *D* is represented by a software element with HUID *H*, then it will *always* be the case that *D* is represented by a software element with HUID *H*.

Consequently, with the HUID, an application is able to look for the FCM of a specific functional component even when the network configuration has changed, and the functional component may be represented by another FCM on another FAV/IAV device with a new HAVi SEID.

Unique and persistent identification of FAVs, IAVs and BAVs (in BAV mode) is possible due to the following facts:

- Each FAV, IAV and BAV device supports IEEE 1394 and IEEE 1212, so the device can be identified uniquely and persistently by the GUID.

- The DCM code unit and the FAV, IAV, or BAV (in BAV mode) device are always from the same manufacturer (or at least manufacturer compliant) as the device itself. So, the device manufacturer can (and has to) take care that each functional component always has the same tag.

However, for LAV devices (or BAV devices in LAV mode) it may not always be possible to identify a functional component on an LAV device persistently for the following reasons:

- The device cannot be uniquely and persistently identified. An LAV does not need to support 1212 (defining the IEEE 1394/1212 GUID) or may even be connected via some other physical bus (e.g. USB or SCART). So, the GUID cannot be used for their unique identification. Although it may be the case that there are other ways to identify the device, it may also be the case that such LAV devices cannot be uniquely identified at all (since two devices can be completely identical and their location or connection point to a bus cannot be determined).

- The device can be uniquely defined, but the DCM representing the device may differ (after network re-configuration). Then, it might not be possible to determine the proper FCM of a specific functional component within a DCM. This identification is the responsibility of the DCM code unit, which is, for an LAV, provided by the IAV/FAV that hosts the LAV. Since the communication between IAV/FAV devices and the LAV device is not (HAVi) standardized, different IAV/FAV manufacturers may select different identifications of FCMs. E.g., for an AV/C-CTS device with a tuner and a VCR, an IAV of Vendor A might provide a DCM in which the FCM of the tuner is identified by tag 1 and the FCM of the VCR by 2, while an IAV of Vendor B has a DCM in which the VCR is identified by 1 and the tuner by 2.

Although it might not always be possible to uniquely and persistently identify FCMs of functional components of LAV devices in a general way, there are many situations in which it can be done. For example:

- Assume a network with an IAV and one LAV that can be identified uniquely and persistently. Since the IAV can identify the LAV each time it installs its DCM code unit, it can assure that the same DCM code unit is always installed. So, this specific DCM could be identified by the GUID of the IAV, combined with an identification of the LAV.

- Assume a network with two IAVs and one LAV that can be uniquely identified, all from the same manufacturer. The manufacturer can assure that the DCMs for that LAV are the same on both IAVs and that both DCMs (and FCMs) can be identified by a vendor specific identification.

- Groups of manufacturers making devices for some standard, e.g. AV/C-CTS or CAL, may agree on how these types of devices should be embedded in HAVi. When DCM manufacturers follow this embedding, it can be guaranteed that the DCMs (and FCMs) for these types of devices are similar and obtain the same identification.

So in general, HUIDs for LAVs (and BAVs in LAV mode) are not always unique and persistent. However, HUIDs will exploit as much as possible the situations described above. Each DCM and FCM stores its HUID in the Registry. DCMs and FCMs can then be retrieved uniquely from the Registry. An application can choose to store locally any information, such as attribute values, it retrieves from the Registry. When the application wants to find a DCM/FCM again, it can query the Registry with these attributes and retrieve the SEID of the DCM/FCM.

3.5.1.3 User Preferred Name

To allow a more user friendly reference to a device, it is possible to assign a “user preferred name” to a DCM. This might be something like “John’s VCR”, “The Red Box”, or “The VCR downstairs”. It is a system wide name that the user can rely on for identifying a thing (e.g. the one with the smart card slot) with that name. The name of a DCM can be retrieved via the DCM API, and can also be found in the Registry. The name can be modified via a method of the DCM API. Quite likely a graphical user interface for the DCM would provide a user friendly way to fill in this name. On modification of the name, the DCM is responsible for modification of the Registry entry to keep the naming consistent. For user convenience, it would be best if the DCM could store this name persistently (e.g. in some non-volatile memory on the device), so that the same name can be used over time, even if the device is switched off and on. However, this may not be the case. The user preferred name may be lost in case of a power off, or a DCM reinstallation (after a network reset) and the user would then need to assign the user preferred name again. Consequently, a DCM is allowed to provide the empty string as the user preferred name.

The user preferred name is meant as an aid to the user for identification. Therefore, the consistency of the usage is also in hands of the user, and HAVi does not provide any means to check or guarantee these consistencies.

3.5.1.4 Native Commands

HAVi has been designed to support the embedding of non-HAVi devices in the HAVi framework. To allow extensive control of these devices, HAVi provides a way to pass native commands to the device. This could be a command native to that specific device, native to the vendor, or native to other non-HAVi standards (e.g. CAL, AV/C). A native command may have side-effects on the standard HAVi DCM service or the service of one of its FCMs. It is the responsibility of the DCM (and its FCMs) to assure that the HAVi standard interface is not violated and to determine whether this specific native command is accepted or not. E.g., a DCM may receive a AV/C “play” request for a VCR sub-unit that is controlled by one of the FCMs. It is the decision of the DCM as to whether the native command is executed when the FCM has been exclusively reserved by another application.

3.5.1.5 Connection Management

DCMs also provide a high level API to allow other objects to query the state of connections within the device and to manipulate those connections. This API is used by the Stream Manager. The connection management part of the API allows device connections to be established, both internally between functional components, and from functional components to the external network. Connection status can be queried and connection capabilities (transmission formats) can also be queried.

A user may directly manipulate a device through a Level 1 or Level 2 interaction. If such an interaction starts, it is possible that connections within the device already exist. In that case, the interaction should normally keep these connections in place. Any DCM or device activity generated by the user interaction may result in certain streaming behavior through these connections. If no such connections exist, but are required at some time, the DCM can establish them, possibly through interaction with the user.

3.5.1.6 Level 1 User Interaction

A DCM may provide a Level 1 device specific user interface via a dedicated set of API primitives as described in section 5.12 on APIs for Data Driven Interaction. The DCM indicates in the Registry whether this form of user interaction is provided.

3.5.1.7 Level 2 User Interaction

A DCM may also support a Level 2 solution by providing a havlet. FAVs can upload, from the DCM, the bytecode for the havlet. A DCM that provides a havlet indicates this capability in the Registry.

3.5.1.8 Resource Management

DCMs can be involved in reservations and scheduled actions. Section 3.8 on Resource Management describes the consequences of this involvement for DCMs and FCMs. A DCM has to take care of “dependent FCMs” in the same DCM code unit that are bound to other FCMs for some reason, and that must always be “internally reserved and released” together with those FCMs.

If a DCM also supports the DDI protocol, it is the responsibility of the DCM acting as a DDI Target to ensure that a `DdiTarget::UserAction` message does not result in reservation violations. This can be accomplished, for instance, by the DDI Target not "showing" facilities (DDI elements) that are reserved by other applications. It also indicates that a reservation is not considered as "part of" the `DdiTarget::Subscribe`. It is up to the application whether or not it reserves one or more (or all) FCMs before the subscription. It should be noticed that "unreserved" FCMs are free for use by any application and thus also via a DDI Controller. If reservations are done by the subscribing application, the DDI Target of course should take them into account for allowing `UserActions` of that application.

3.5.2 Functional Component Modules

An FCM is a software abstraction of a functional component providing the functionality of that functional component to the HAVi software environment and applications. HAVi applications will not communicate with a functional component directly but only through the FCM. An FCM is a HAVi object in the sense that it is registered in the Registry and it can communicate with other HAVi objects via the HAVi Messaging System.

HAVi has defined the following FCMs: tuner, VCR, clock, camera, AV disc, amplifier, display, AV display, modem, and Web proxy.

3.5.2.1 General

An FCM provides a set of basic methods for device control and observation. This API can be used by HAVi system components and applications. It includes the following functionality:

- HUID information – Provides the HUID of the FCM as well as the corresponding DCM.
- Type information – Within HAVi a set of functional component types has been identified, e.g. VCR, tuner, display, etc. For each HAVi defined type there is a HAVi defined message set specifying an API for the control and observation of such a device. This API gives the type of the functional component represented by this FCM and indicates which HAVi messages are supported by this FCM.
- Power management – Provides means to change the power state of the functional component from "on" to "stand-by".
- Native Commands – Provides means to control the device in its native command set (CAL, AV/C, etc) and is similar to the interface on the DCM.

3.5.2.2 Notifications

For certain events, software elements may like to subscribe directly to the event source since it knows from which software element the event will be generated and it is only interested in an event from that source. In the case of a VCR for example, when an application subscribes to an "end-of-tape" event, it knows exactly from which VCR the event is expected. It is not interested in receiving end-of-tape" events from a VCR it is not controlling.

Moreover, for some types of events, the event subscription must be flexible enough to allow conditional event programming: in a large number of cases, it may be interesting to subscribe to a conditional event (i.e., an event is generated if the condition is matched). Generally, functional components have a certain number of state variables that they make available via their API. It

would be restrictive to define *a priori* which of these variables shall be able to generate events, and which not. Moreover, it would be nearly impossible to define which state transitions may be interesting for future applications, and which not. So, proposing a special event for each imaginable combination would result in a huge and unwieldy API specification.

For this reason, a general mechanism, called “notifications”, is provided. Two methods of the FCM API allow an application to subscribe and unsubscribe to notification of specific changes of a specific attribute. By a relational expression (“attribute is bigger than 10”), the application can indicate what kind of change it is interested in.

The general FCM API only provides the general part of this API, i.e. the methods for subscribing and unsubscribing. It does not provide the specific sets of attributes. Those should be provided by the specific functional component APIs.

Although the mechanism of notifications is described as a part of an API for an FCM, this mechanism could also be a part of the API of any software element capable of generating events.

3.5.2.3 Connection Management

FCMs also provide a high level API to allow other objects to query the state of connection plugs on the FCM that may be involved in internal or external connections. This API is used by the Stream Manager. The connection management part of the API allows queries about connection status and connection capabilities (stream types).

3.5.2.4 Resource Management

FCMs can be involved in reservations and scheduled actions. Section 3.8 on Resource Management describes the consequences of this involvement for DCMs and FCMs. An FCM is not obliged to support reservations if it only has non-state changing methods. In this case, a “not supported” return value will be returned if an application tries to reserve it (and possibly for other reservation-related methods). However, each FCM with state-changing methods shall support reservation facilities.

3.5.2.5 Virtual FCMs

HAVi FCMs typically are interfaces to functionality provided directly by hardware. However it is also possible for an FCM to encapsulate software functionality. This is similar to the idea of a *virtual device*, a device realized, at least partially, by software mechanisms. Based on this analogy, the following two categories of FCM are defined:

- *physical FCM* – an FCM which controls the operation of a functional component of a target device. When a HAVi message is sent to a physical FCM it results in implementation dependent communication between the FCM (or its DCM) and the target device.

- *virtual FCM* – an FCM which controls the operation of software-based processes. When a HAVi message is sent to a virtual FCM, the message is processed internally by the FCM and does not necessarily involve communication with other devices on the network.

FCMs in general present both a control interface (the set of HAVi messages to which they respond) and a content interface (a set of plugs). In order to allow virtual FCMs the same range of functionality as physical FCMs, they must be capable of presenting both control and content interfaces.

3.5.3 Havlets

A havlet is a Level 2, device-dependent user application. A havlet is typically a proprietary application that offers a user interface for the control of a specific target device. For the actual control of the device, the havlet makes use of the DCM of the target device. A havlet is a HAVi object in the sense that it is registered in the Registry and can communicate with other HAVi objects via the HAVi Messaging System. Since a havlet is a Level 2 concept, a havlet only runs on an FAV device (because that FAV device has uploaded and installed the havlet code unit). A havlet offers an interface to the user on the FAV device where it resides.

3.6 Device Control Module Manager

The DCM management system is responsible for installing and uninstalling DCM code units for the control of BAV and LAV devices that are directly connected to a HAVi 1394 network. DCM management is collectively performed by the DCM Managers on FAV and IAV devices. DCM code units for FAV and IAV devices are managed by the devices themselves in a proprietary manner. Non-1394 devices shall be managed by FAV or IAV devices in a proprietary manner.

Each FAV device has a DCM Manager, but for IAV devices a DCM Manager is optional. An IAV device is not required to participate in the DCM management process if it will not host any BAV or LAV device on the HAVi 1394 network. (It shall be indicated in an IAV's SDD whether it has a DCM Manager or not – see section 9.10.4.2.)

In the sequel, a BAV or LAV device will be termed a *guest*. Each guest shall be facilitated by a DCM code unit installed on an FAV or IAV device, termed a *host*.

A DCM code unit may be a Java code unit to be loaded and installed on an FAV device, or a native code unit to be installed on an FAV or IAV device. Installation of a DCM code unit results in the installation of DCM and FCMs which shall register themselves.

Each DCM Java code unit is accompanied by a *profile*, which includes the values of the transfer and installation size parameters needed by the code unit and its components (see section 9.10.7). The data in the profile are needed by a host to determine whether it can upload and install the corresponding DCM Java code unit.

Each DCM Manager offers a number of methods that can be invoked by software elements, and a number of global events that notify the results of DCM code unit installation and uninstallation. Most DCM management activities are triggered by a network reset event, which is typically generated when the network topology changes or a device is (de)activated. The network topology changes if, e.g., a device is added to or removed from the network, if the network is split, or if two networks are joined.

3.6.1 DCM Code Unit Installation and Uninstallation

- The actions of the DCM management system as the result of a network reset event are discussed next. These actions can be fine tuned or overruled by *preferences*, which can be set locally on certain DCM Managers (discussed in section 3.6.2).

- For each guest there shall be at most one installed DCM code unit in the entire network. Due to resource limitations, it may occur that no DCM code unit is installed for some guest on the network. It may also occur that a DCM code unit uninstalls itself for some reason. Normally, another host will be selected for a guest’s DCM code unit if the previous host is removed from the network.

- A DCM code unit shall be uninstalled if the corresponding guest is no longer available on the network.

If a DCM code unit is to be installed, it shall be according to the following scheme (assuming no preferences are set):

- If an uploadable DCM Java code unit for the guest is available, an FAV host shall be selected to load and install the code unit. If this fails, any host that can install an embedded DCM code unit for the guest in a proprietary way shall be selected.

- A BAV device may internally store an uploadable DCM Java code unit and/or a URL for such a unit in its SDD data. If a URL for an uploadable DCM Java code unit is specified in a BAV device, it shall be loaded from the specified location and installed on an FAV host, instead of the DCM code unit contained in the BAV device. If this fails, the DCM code unit contained in the BAV device shall be installed instead. (It is recommended for BAV devices to contain a DCM code unit. If there is no code unit, a URL for a code unit should be contained).

- All other conditions being equal, a host with the lowest number of installed DCM code units is selected.

- If a host rejects the DCM, either before or after installation is attempted, the next most suitable candidate is attempted until all possible candidates have been tried.

A URL is specified as `<scheme>://<host>/<path>`. The scheme can be, for example, `http` or `ftp` for IP-based protocols, or `file` for storage media. For URLs in the configuration ROM of BAV devices the `file` scheme shall not be used. The URL shall denote an uploadable DCM Java code unit and/or its profile on the Internet or on a (persistent) storage medium. The validity of URL syntaxes is outside the scope of this document. Note that DCM Managers need not have knowledge of URL syntax and semantics.

A DCM Java code unit and its profile are two separate files, each with their own file name. The following convention applies:

- DCM Java code units shall have a file name with extension `hdc` (HAVi DCM Code).

- DCM Java code unit profiles shall have a file name with extension `hdp` (HAVi DCM Profile). The format of a “`.hdp`” file conforms to section 9.10.7.

DCM Java code unit URLs shall be specified without the extensions “`.hdc`” or “`.hdp`”. The proper extension will only be added when either a profile or a code unit is actually retrieved (see also section 9.10.8).

The DCM management protocol will exploit the URL access capability of devices. A host device may decide to cache a loaded DCM code unit instead of retrieving it from the URL-designated location. DCM Managers shall be able to use URL access capable FCMs in the network, but they can also offer URL access capability in a proprietary manner.

Uploadable DCM Java code units shall be encoded in the format described in section 7.4.1. This applies both to code units contained in BAV devices and those designated by a URL.

3.6.2 Preferences

The activities of the DCM management system are guided by preferences, which can be set by applications. Preferences typically specify deviations from the default installation actions described in the previous section. Methods are available for setting and retrieving preferences on DCM Managers in the system. Preferences are optional; DCM management will function without any set preference. If a preference is set on some DCM Manager, it shall be stored persistently if possible, so that it does not have to be entered each time a device is powered up. The value of a preference that is not set on a host is *unspecified*. Preferences shall be modifiable by the user.

The first three preferences in the following table are associated with either a single guest (by its Global Unique Identifier, or GUID) or with a guest model (by its Vendor Model Identifier, or VMID – see section 5.8.2). For an LAV guest, only the GUID variant is possible, since a VMID is not available for LAV devices. A GUID-based preference for a guest overrules a VMID-based preference for the guest model, without causing a conflict. The fourth preference is not associated with a GUID or VMID.

Table 8. DCM Installation Preferences

Preference	Type
DCM_PREFER_VENDOR_HOST	boolean
DCM_PREFERRED_HOST	GUID
DCM_PREFERRED_URL	string
DM_PREFERRED_URL_DEVICE	GUID

- DCM_PREFER_VENDOR_HOST – Designates whether a host of the same vendor as the guest is preferred for installing a DCM code unit. If for any guest or guest model the value is set to True by any DCM Manager, the DCM management system shall give preference to a host of the same vendor. The value used for DCM code unit installation is determined by taking the disjunction (logical OR) of all values from the DCM Managers. If unspecified by a DCM Manager, the value False is assumed. However, a vendor may return a value of True for any of its own guests if the value is unspecified.

- DCM_PREFERRED_HOST – Designates a specific host that is preferred for installing a DCM code unit for some guest or guest model. The reason for setting this may be performance or reliability considerations.

- DCM_PREFERRED_URL – A URL designating the location of an uploadable DCM Java code unit and its profile for a guest or guest model. For a BAV guest, this preference is used to specify a DCM code unit to be installed instead of a DCM code unit designated by a URL in the BAV device or the DCM code unit contained in the BAV device. For an LAV guest, it is used to specify an uploadable DCM Java code unit that can be installed on an FAV host. (Note that an uploadable DCM Java code unit cannot be installed for LAV guests for which this preference is not set.)

- `DM_PREFERRED_URL_DEVICE` – Designates a specific host or guest that is preferred for URL access to retrieve DCM Java code units. For example, the user may designate a device capable of Internet access or a device that stores DCM code units. If this preference specifies a host, it should be used to retrieve URL-designated code units. If this preference specifies a guest, it indicates that a DCM code unit should be installed for this guest before installing DCM code units for any other guest. The guest should subsequently be used to retrieve URL-designated code units. Note that the URL access capability of a device that is not specified by `DM_PREFERRED_URL_DEVICE` may be ignored.

For any preference, it may occur that conflicting values have been set at different DCM Managers. One of the values will be selected arbitrarily. However, for `DM_PREFERRED_URL_DEVICE` an FAV device takes preference over an IAV device, and a host takes preference over a guest. Only devices that exist in the network are taken into account for this preference. DCM-related preference conflicts are reported in `DcmInstallIndication` events.

Preferences relating to some guest only take effect when a DCM code unit is to be installed for that guest, e.g., upon network reset events, and upon invoking (un)installation requests. Setting (or changing) preferences relating to some guest does not result in the DCM code unit installation, or re-installation even when a DCM code unit for that guest has already been installed. There is a priority among the preferences to resolve ambiguities. For installing a DCM code unit the following steps are taken, in the order listed. The DCM manager will execute each step in turn, attempting to install the DCM code unit on all hosts valid under the current step, until the DCM code unit is installed successfully on a host, or all the steps listed have been attempted.

- If <preferred host> is specified and is an FAV device, and <preferred URL> is specified, try to install the code unit designated by the preferred URL on the preferred host.
- If <preferred host> is specified and is an FAV device, and the guest is a BAV device, try to install the code unit designated by the SDD URL in the guest on the preferred host.
- If <preferred host> is specified and is an FAV device, and the guest is a BAV device, try to install the code unit contained in the guest on the preferred host.
- If <preferred host> is specified and is an IAV or FAV device, try to install a code unit on the preferred host in a proprietary manner.
- If <prefer vendor host> is `True`, try to install a code unit on a host of the same vendor as the guest in a proprietary manner.
- If <preferred URL> is specified, try to install the code unit designated by it on any FAV host.
- If the guest is a BAV device, try to install the code unit designated by the SDD URL in the guest on an FAV host.
- If the guest is a BAV device, try to install the code unit contained in the guest on an FAV host.
- Try to install a proprietary code unit on any host.

In case an undesirable DCM code unit installation was performed it is possible to use the DCM Manager methods for explicit uninstallation and subsequent installation of code units. For example,

the home network configuration at some time during the network start-up may be incomplete, leading to premature code unit installations.

3.6.3 Interaction between DCM Code Unit and DCM Manager

A DCM Manager shall only install and uninstall DCM code units on the local device. Once loaded (if applicable), the DCM Manager controls the DCM code unit through Java DCM code unit methods. Each loaded DCM code unit shall offer these methods to the DCM Manager. However, the vendor of a HAVi host device can arrange for interaction between a DCM Manager and an embedded DCM code unit to take place in a proprietary manner.

The DCM code unit method install(nodeId, listener) is invoked by the DCM Manager to install the DCM code unit; it is the first method a DCM code unit shall accept. The DCM code unit components (DCM and FCMs) shall be installed and registered. The DCM code unit method uninstall() is invoked by the DCM Manager to uninstall a DCM code unit if required; it is the last method a DCM code unit shall accept.

After a DCM code unit has uninstalled itself, it signals this to the local DCM Manager through the Java listener method uninstalled(). This call shall only occur after all DCM code unit components have been uninstalled and unregistered. After this call, a new DCM code unit can be installed for the guest, if required. A DCM code unit shall uninstall itself if the DCM manager requests it to do so. However, it can also do this on its own initiative.

The DCM code unit concepts and methods are discussed in section 7.4.1.

3.7 Stream Manager

3.7.1 Objectives

The Stream Manager provides an easy to use API for configuring end-to-end isochronous (“streaming”) connections. Connections may be point-to-point or utilize “broadcast” sources or sinks. The responsibilities of the Stream Manager include:

- configuration of both *internal* connections (within a device) and *external* connections (between devices)
- requesting and releasing transport system resources
- providing global connection information
- support of plug compatibility checking
- support for connection restoration after network resets

3.7.2 Design Decisions

- A Stream Manager runs on each FAV device; implementation on IAV devices is required only if applications on the IAV need Stream Manager services.
- A Stream Manager provides connection creation services only to applications running on the same device as the Stream Manager itself.

- The Stream Manager API is based on the IEC 61883 plug model and the HAVi functional component model.

- IEC 61883 broadcast and point-to-point connections are supported. (*Note,* due to the restrictions specified in 61883.1 on broadcast connections, Stream Managers may not be able to establish broadcast connections for some devices, even though the devices themselves support broadcast connections.)

- Connections do not cross transport boundaries (i.e, connections have a single "transport type" – see below).

- Connections originate and terminate at functional components (rather than devices).

- Plugs should satisfy stream type compatibility, i.e. the Stream Manager leaves to the application the problem of configuring stream format converters (functional components which sink a stream of one type and then source a stream of a second type).

- The Stream Manager is responsible for requesting and releasing isochronous resources, this includes: IEC 61883 PCRs (plug control registers), 1394 bandwidth and 1394 channels.

- A standard naming scheme is used for stream types, transport types and transmission formats. The stream type and transport type naming schemes are specified by HAVi and, for stream types, the naming scheme may be extended by vendors; transmission formats are transport type dependent and specified outside of HAVi.

- Implementations of the Stream Manager are required to make efficient use of 1394 resources by using the IEC 61883 "overlay" mechanism whenever possible.

3.7.3 Definitions

connection – a uni-directional data transfer path created by a Stream Manager. Typically used for streaming content. Connections originate and/or terminate at functional components. A connection is either an *internal connection* or an *external connection*. *Non-HAVi connections* refer to data transfer paths created by non-HAVi applications or devices.

internal connection – a connection where data is transferred within a device.

external connection – a connection where data is transferred across a device boundary.

attachment – The segment of a connection which exists between a DCM plug and the plug of one of its FCMs.

device connection – an internal connection or an attachment.

connection identifier – Stream Managers and applications refer to connections via unique identifiers. These values persist throughout the lifetime of the connection. Connection identifiers for connections created by Stream Managers are globally unique and their reuse should be avoided as much as possible.

stream type – identifies a media representation, this may be a data format (for digital media) or signal format (for analog media), e.g. CD audio, composite video.

transport type – identifies a transport system. HAVi compliant implementations of the Stream Manager must support three transport types: `IEC61883` for IEC 61883 connections running over 1394, `INTERNAL` for connections internal to a device, and `CABLE` for connections over external cabling. Support for other transport types may be added to future versions of the HAVi Stream Manager, or may be provided by proprietary extensions to the Stream Manager.

transmission format – identifies the transmission protocol used to send a stream type over a transport type. For `IEC61883`, the transmission format corresponds to the `FMT` and `FDF` fields in the Common Isochronous Packet (CIP) header. For `CABLE`, the transmission format identifies forms of signaling used on physical cables. HAVi assigns a 16-bit code to many of the signal formats commonly used by CE equipment, these are listed in Annex 11.12. For `INTERNAL`, transmission formats are not used.

plug – a resource, provided by a device and used to build a connection. A plug is, at least conceptually, the source or sink of the data carried by a connection. There are two main plug categories: *functional component plugs* and *device plugs*. A device plug is either a PCR or an external cable plug. Functional component plugs and device plugs are represented by FCM plugs and DCM plugs respectively. However it should be noted that FCM and DCM plugs are software abstractions and that when one says, for example, two FCM plugs are connected, this implies that the underlying functional component plugs are connected. This document also refers to *source plugs* and *sink plugs*, and *output plugs* and *input plugs*.

channel – a resource, provided by a transport mechanism (e.g., IEEE 1394) and used to build a connection.

3.7.4 Streams

The concept of *stream* as used by the Stream Manager is based on the isochronous data flow concept of IEC 61883 with two differences: The first difference is that a IEC 61883 data flow starts at a device source plug and ends at a device sink plug – while a stream typically starts at a functional component source plug, goes to a device source plug, then a device sink plug, and ends at a functional component sink plug. The second difference is that streams are typed, i.e. for each stream there is an associated *stream type* identifying data representation, bandwidth, and other attributes of the stream.

The following summarizes the properties of streams, channels and connections:

- a stream is associated with a single source and a set of connections and their channels
- device plugs can be *bound* to channels
- channels are either *fully bound* (bound to a source device plug and a sink device plug) or *partially bound* (bound to a source device plug only or to sink device plugs only)
- a channel can be bound to zero or one source device plugs and zero or more sink device plugs
- a sink device plug can be bound to at most one channel
- a source device plug can be bound to at most one channel
- a channel may have no source device plug, but no data will flow over the channel until a source device plug has been bound

3.7.5 Connections

3.7.5.1 Device Connections

A device connection is one of:

- An internal connection from a functional component source plug to a functional component sink plug.
- An attachment from a functional component source plug to a device source plug.
- An attachment from a device sink plug to a functional component sink plug.

The following rules apply to device connections.

- Functional component sink plugs can have at most one device connection.
- Functional component source plugs can have many device connections.
- Device source plugs can have at most one device connection.
- Device sink plugs can have many device connections.

Stream Managers establish and break device connections via the `Dcm::Connect` and `Dcm::Disconnect` APIs. Stream Managers shall request the DCM to connect even if the device connection already exists, this allows the DCM to "overlay" device connections and maintain information about the usage of device connections.

3.7.5.2 Internal Connections

Internal connections are a form of a device connection. Consequently:

- Functional component sink plugs can have at most one internal connection.
- Functional component source plugs can have many internal connections.

Stream Managers establish and break internal connections via `Dcm::Connect` and `Dcm::Disconnect`.

3.7.5.3 External Connections

An external connection will involve attaching functional component plugs to IEC 61883 plug control registers (transport type is `IEC61883`) or to external device plugs (transport type is `CABLE`). External device plugs are connected, typically, by physical cabling. It is not required that the HAVi Stream Manager be aware of the configuration of such cabling. If the Stream Manager is asked to establish a connection for which successful operation would require physical cabling (e.g., analog audio or video cables), the Stream Manager assumes the user has made the proper connections.

3.7.5.4 Global Connection Map

The Stream Manager is capable of constructing a map of all connections within the home network

established by HAVi applications. The Stream Manager does not guarantee that the map is built in one atomic operation.

3.7.5.5 Connection Examples

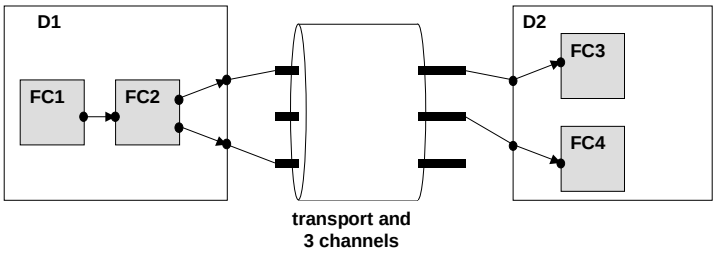

Figure 20. Connection Diagram

The diagram above shows several examples of connections. Here *D1* and *D2* are devices and the *FCi* are their functional components, the small dark circles represent plugs. There are four connections:

- an internal connection between *FC1* and *FC2*
- an external connection between *FC2* and *FC3*
- an external connection originating at *FC2* (the channel is partially bound)
- an external connection terminating at *FC4* (the channel is partially bound)

Note that, for example, the connection between *FC2* and *FC3* involves two device connections (attachment of the *FC2* source plug to the *D1* source plug, attachment of the *D2* sink plug to the *FC3* sink plug) and the binding of the channel to the device source and sink plugs.

Note that for FCMs with source plugs and sink plugs, for example FC2, it is not guaranteed that such an FCM will or will not pass through the signal from a sink plug to a source plug.

3.7.6 Transport Types

Transport types are represented by 16-bit identifiers. The values assigned to the three transport types, `CABLE`, `INTERNAL` and `IEC61883`, are listed in 11.14.

3.7.7 Stream Types

Stream types are represented by a stream type identifier – a data structure containing an IEEE 1394 Vendor ID field and a type number field. The Vendor ID of value 0x0 is reserved for use with HAVi defined stream types. Stream types form a hierarchy, the HAVi stream types are arranged in

the hierarchy shown below (only the structure of the hierarchy and some representative stream types are shown, for a complete list of stream types see Annex 11.11):

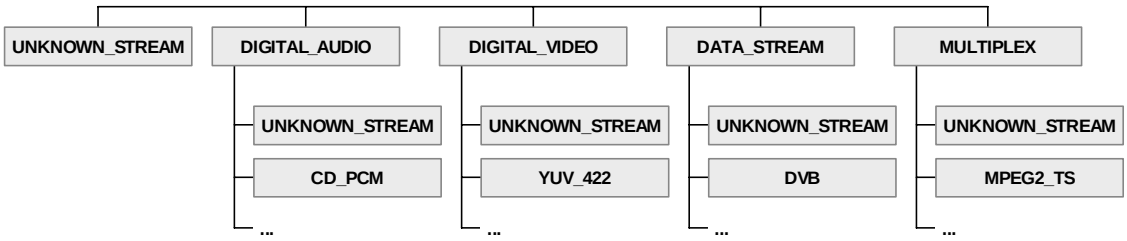

Figure 21. Stream Types

The stream types `UNKNOWN_STREAM`, `DIGITAL_AUDIO__UNKNOWN_STREAM` etc. are used for streams of unknown type. During connection establishment, the Stream Manager reduces stream type compatibility checks for such plugs. Instead it follows a "connect and test" procedure (i.e., it attempts to connect and then tests for success or failure).

HAVi reserves `VendorId` = 0 for stream types. Device manufacturers can define their own stream type hierarchy using their vendor id. Stream types with different vendor id will not match each other except for the UNKNOWN case (see 5.9.5.1 Stream Type Matching).

3.7.8 Plug Compatibility Checking

When the Stream Manager attempts to connect plugs it tests whether:

- transport type is compatible (i.e., a common transport type can be determined)
- directionality is compatible (i.e., output plug to input plug)
- stream type is compatible (i.e., the stream type of the source and sink plugs match as described in section 5.9.5.1)
- bandwidth is compatible (i.e., the required bandwidth of the source is less than or equal to the bandwidth which the sink is capable of consuming)
- transmission format is compatible (i.e., the transmission format of the source and sink plugs match as described in section 5.9.5.2)

If any of these tests fail then the connection cannot be established. The compatibility check about stream type, bandwidth and transmission format is not applied to the `CABLE` and `INTERNAL` connection.

3.7.9 Connection Restoration: Network Reset

After a network reset, the IEC 61883 procedures for restoring connections are invoked by the Stream Manager for each connection (of the `IEC61883` transport type) that it has created. During connection restoration the Stream Manager builds a new list of connections for which it is responsible and may post `ConnectionDropped` events. This event is posted when:

- as a result of a network merge there is insufficient bandwidth available for a connection,

- as a result of a network partition the source or sink of a stream has been lost.

Stream Managers shall complete IEC 61883 connection restoration during the first 1 second after network reset (this is the “isoch_resource_delay” period specified by IEEE 1394-1995). Also, the Stream Managers shall indicate to DCMs the device connections they wish removed by using `Dcm::Disconnect`.

3.7.10 Connection Restoration: Power Off

If a device running a Stream Manager powers down then during handling of the subsequent network reset the connections created by the Stream Manager will not be restored. However, since a Stream Manager only creates connections for local applications then the applications themselves will no longer be running so there should be no need for restoration.

3.7.11 Connection Dropped

A connection is dropped under the following circumstances:

- the Stream Manager which created the connection is requested to drop the connection
- the owner of the connection or an FCM for the source or sink leaves
- the Stream Manager receives a `PowerStateChanged` event indicating loss of power for the source or sink
- the source or the sink is no longer present after a network reset (as described above)
- the Stream Manager fails to restore the connection after a network reset (as described above)
- the Stream Manager detects a removal of an device connection (via the DCM `DeviceConnectionDropped` event) and determines that a connection the Stream Manager has established is no longer operable

3.7.12 Connection Changed

Stream Managers subscribe to `TransmissionFormatChanged` and `StreamTypeChanged` so that they can maintain a consistent view of the transmission format and stream type used by a connection. Stream Managers subscribe to `BandwidthRequirementChanged`, in order to be informed of changes (or attempted changes) in bandwidth associated with a connection. Finally Stream Managers subscribe to `DeviceConnectionDropped` and `DeviceConnectionChanged` in order to respond to changes in internal configuration of the source or sink of a connection.

3.7.13 Connection Establishment and Drop Order

Stream Manager establish connections of transport type `IEC61883` by configuring from source to sink (i.e., first connect source attachment, set source stream type and transmission format, update the `IEC61883` plug control registers, connect sink attachment, then set sink stream type and transmission format). Oppositely, Stream Manager should drop connections of transport type `IEC61883` from sink to source.

If Stream Manager fails DCM/FCM API or set IEC61833 plug control register to establish a connection, Stream Manager does not need to restore these settings to the previous state, but shall restore the settings of isochronous resources and plug control registers according to IEC61883 CMP.

3.7.14 Connection Overlay

HAVi connections can be overlayed not only for IEC61883 transport type, but also for `CABLE` and `INTERNAL` transport types. Overlay of device connection part of HAVi connections are managed by DCMs and Stream Managers. DCM maintains a list of Stream Managers that have device connection on the DCM. Once a Stream Manager calls `Dcm::Connect`, the Stream Manager is memorized in the list in the DCM. The DCM will not maintain how many device connection is owned by each Stream Manager and each Stream Manager shall maintain the count by itself. (Note: Stream Managers should call the `Dcm::connect` for each time making overlay for reservation protection check reason.)

When an SE calls `StreamManager::Drop`, Stream Manager decrements the counter but does not call `Dcm::Disconnect` until no connection on the attach remains. On executing the last drop process, the Stream Manager calls `Dcm::Disconnect`, the DCM erases the Stream Manager from the list. In this way, overlay within a Stream Manager and overlay between Stream Managers are maintained.

In case of overlays, when the source and/or sink attachment already have the desired stream type the Stream Manager should not call `Dcm::SetStreamTypeId` again. Similarly, in case of overlays, when the source and/or sink DCM already have the desired transmission format the Stream Manager should not call `Dcm::SetTransmissionFormat` again.

3.8 Resource Manager

Applications in the network will typically use a set of FCMs to perform a task on behalf of one or more users. FCMs are called *device resources* in this context. Usually, also *network resources* are involved in resource management, since these serve to create useful collaborations in audio/video streaming between DCMs and FCMs in a HAVi network. 1394 bandwidth and channel numbers are such network resources. The Resource Manager only deals with device resources.

The Stream Manager handles network resources on behalf of clients. Connection and bandwidth management will typically be requested by applications after the involved device resources have been reserved for usage. In the following, resources stands for device resources, or FCMs.

Resource management serves to guide software elements competing for and using the set of resources in the network. Such software elements are called *clients* in this context. A group of device resources will be reserved by a client in an all-or-nothing fashion. The resource management system serves to ensure that clients that have reserved resources can rely on not being disturbed by other applications that (try to) use these resources. Potential clients that want to reserve resources, but have not yet done so are called contending clients or *contenders*.

There is a Resource Manager on each FAV and IAV device that hosts or can host at least one DCM. The **HAVi_Resource_Manager** SDD field of an IAV device reveals whether a Resource Manager is present.

3.8.1 Resource Reservation

Each Resource Manager offers methods to applications for reserving and releasing resources, as well arranging *scheduled actions*. A scheduled action is a reservation and usage of a set of resources during some future time period. Conflicts in schedules are detected and reported. The resource management system will perform a number of validations for scheduled actions at scheduling time.

The resource management model supports a reservation mechanism. Reservation is used to protect against *control commands* ("write access"), changing the state of a resource. Reservation is not needed for *observation commands* ("read access"). An FCM is not obliged to support reservations if it only has methods that do not change its state. In this case, FCM methods and events related to resource management need not be implemented. Other FCMs shall support reservation facilities.

In general, applications can subscribe to resource events. Such event subscriptions are considered not to change the state of the resource, and therefore belong to the observation category. The FCMs and Resource Managers offer a number of events that can be subscribed to by interested applications to learn about the status of resources with respect to reservations.

Two client roles are distinguished and are relevant in *negotiations* for resources (discussed later):

- *User* – A client that is able to take part in a resource negotiation with a contender. Negotiation will typically involve the human user of the client (to give him or her the opportunity to accept or reject a negotiation). However, it is not required that a human user be involved for a client to take this role.

- *System* – A client that is not able to take part in a resource negotiation with a contender. This will typically be the case if no human user is involved with the client.

A limitation of a system contender is that it can or should not preempt resources from user clients that reject its preemption request. Resource Managers performing a scheduled action (Action Schedulers) are such system contenders. An application is free to act as a user or system contender, although it should choose the correct role in a cooperative situation. User contenders can always preempt resources from any client.

Note that, for instance, an emergency application would typically be a user contender. Although it may acquire resources by direct preemption, it may negotiate first to learn whether a current client has an even higher priority.

A DDI Controller may control a DCM through the DDI protocol. The DDI Controller is responsible for reserving any FCMs associated with the DCM if it requires exclusive access to the related device or some of its functional components. See section 3.5.1.8 Resource Management.

The resource management model relies on the following principles:

- Clients are cooperative, i.e., they shall respect the reservation mechanism and their user (or system) role.

- Before using control commands, an application will normally reserve the needed resources. It shall release the resources when they are no longer needed.

- With due restrictions, a contender may negotiate through a Resource Manager with other clients to take over (preempt) their reservation of resources. It may also decide to preempt resources without negotiation. This process is described later.

The next figure shows an example of the interaction between clients in a network with FAV or IAV TV and VCR devices. (In this example, only one resource is reserved. In general, a Resource Manager will attempt to reserve a group of resources requested by a contender.)

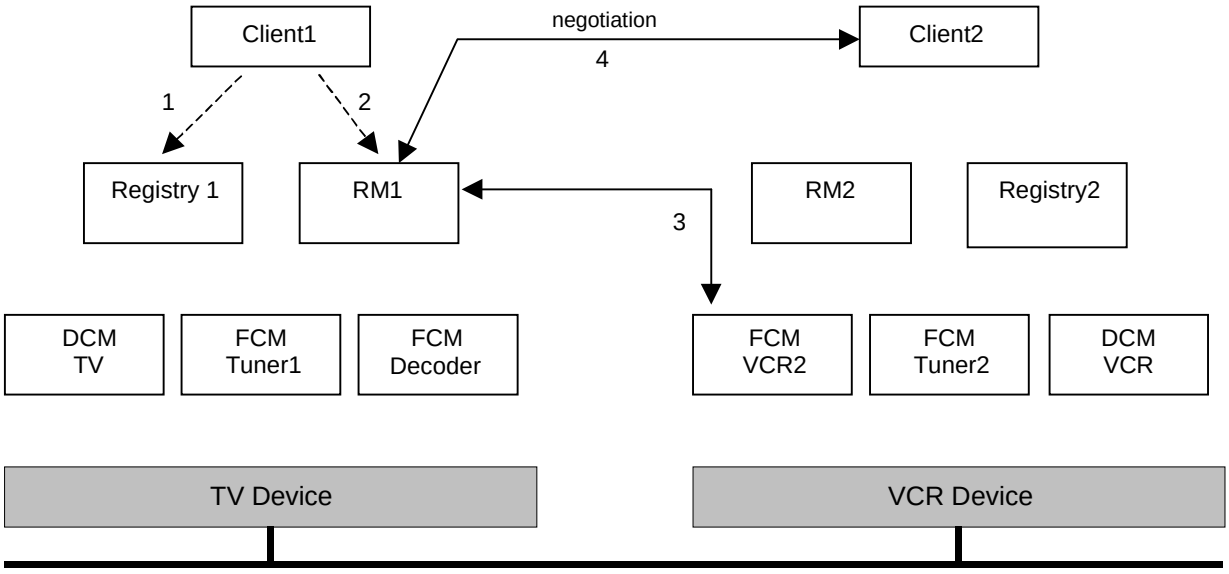

Figure 22. FCM Reservation

Client1 first queries its local registry for VCR resources present in the network (arrow 1) and then calls a (local) Resource Manager (*RM1*) in order to reserve the VCR FCM (arrow 2). Next, *RM1* attempts to reserve *VCR2* (arrow 3).

Assume *VCR2* is currently reserved by *Client2*. Before attempting to reserve it, *Client1* may negotiate through a Resource Manager to preempt the resource from *Client2*. It will then send a negotiation request to, say, *RM1* (arrow 2). *RM1* will send a preemption request to *Client2* and await a preemption reply from it (arrow 4). An acceptance or rejection is forwarded by *RM1* to *Client1*. Depending on the answer, *Client1* may decide to preempt *VCR2* (even if *Client2* rejected).

There are various types of resources. Some will provide function for producing AV contents like Tuner. Some other will provide both function for producing or recording AV contents like VCR or AVDisc. Yet other will provide audio visual interface to the users like Display or Amplifier. HAVi does not specify which resources to be reserved, but clients are recommended to cooperate respecting following suggestion; As for FCMs that provide audio visual interface like Display or Amplifier FCM, the basic usage may be "free to use" model, that means clients will not reserve FCM without special purposes or reasons. When clients reserve these FCMs, the clients have to take into account the side effect of the reservation that might cause inconvenience to the user.

3.8.2 Resource Sharing

Device resources can be reserved by an application for exclusive use, but can also be shared by more than one application. The resource type determines which sharing options are available to applications.

Any application may change the state of a device resource as long as the resource has not been reserved. However, resources shall protect applications from unauthorized access by other applications in case the resource is properly claimed. A resource shall verify for each command whether the invoker is entitled to perform the command. (It shall check the SEID of the invoker

with its locally registered primary and secondary SEIDs for control commands.)

A resource may be shared by more than one client. A DCM provider can choose the extent of sharing that is supported for each FCM, and the restrictions that apply to sharing clients with respect to each other. Two types of access rights are distinguished:

- *Primary access* – Full control of the resource without any restriction. There is at most one primary user per resource.

- *Secondary access* – Limited control of the resource. The primary client and other secondary clients should never be hampered in their access rights by a secondary client.

The number of secondary clients allowed is determined by the DCM manufacturer. If a primary client releases the resource, a secondary client (if any) will not automatically become the primary client. It may, if desired, attempt to reserve the resource as a primary client. If successful, it will become a primary client, and loses its secondary client position.

As an example, consider a tuner FCM able to operate in one of several multiplexes, each of which supports a number of channels. The primary client can change the multiplex of the tuner, but a secondary client may not. The secondary client can only tune to a channel that happens to be in the selected multiplex.

Note that any application can use observation commands of a resource at any time. It need not reserve the resource for such commands. Also note that user and system clients may accept to operate with either primary or secondary access rights. However, negotiation and a subsequent preemption (discussed in the next section) only apply for primary access rights.

3.8.3 Resource Negotiation and Preemption

A resource is normally used by an application between the moments it chooses to reserve and release the resource. However, there are means for competing applications to hand over resource access rights.

Reserving and releasing resources is always done by a Resource Manager selected by the contender using the methods `ResourceManager::Reserve` and `ResourceManager::Release`. Only Resource Managers are allowed to directly reserve and release resources through `Fcm::Reserve` and `Fcm::Release`.

A user contender can always become the new client of a group of resources by preempting them from their current clients. However, in normal circumstances, a contender will first try a so-called *non-intrusive reservation*. This is a reservation attempt that does not involve any current client. If all resources in the requested group can be acquired in this way, they will be acquired; otherwise, none of them will.

A contender can negotiate with primary clients through a Resource Manager in order to learn whether they are willing to give up their resource reservations. The method `ResourceManager::Negotiate` is used by the contender. The Resource Manager will then invoke the method `<Client>::PreemptionRequest` of all primary clients to forward the request. Each client should respond within a specified timeout with an acceptance or rejection of the request, and an information string. (`PreemptionRequest` should be implemented by applications that take on the user role). Cooperative controllers are advised to follow the following rules:

- If all clients accept the preemption request conveyed in the negotiation, the contender may preempt their resources.

- A system contender shall not preempt resources from clients that have rejected the preemption request.

- To all clients that have received a preemption request, the contender shall send a withdrawal notification if it decides not to preempt their resources (through `ResourceManager::Negotiate`).

So, normally negotiation precedes preemption. The client should respond within a specified negotiation timeout. This negotiation may involve a person behind a current primary user client to agree or disagree with a preemption. However, even if a client disagrees, a user contender can directly preempt its resources. This prevents any application in the network from monopolizing resources. If a negotiation times out, the contender may decide not to preempt the resources of that client, although this is up to its own discretion. This scheme is thought to suffice for home networks, where applications should cooperate in the usage of resources.

Negotiation, preemption, and reservation are always done for a resource group, although a client can choose any subset of resources to reserve or release, even after it has already reserved other resources.

The following scheme describes situations where preemption is recommended or allowed after negotiations between a contender and a set of clients.

preemption after negotiation	user contender	system contender
current user client	upon accepted negotiated reservation requests (preemption always allowed)	upon accepted negotiated reservation requests (else preemption not allowed)
current system client	negotiated reservation requests not accepted (preemption always allowed)	negotiated reservation requests not accepted (preemption not allowed)

For system contenders, preemption is *only* allowed if the current client is a user client, and has accepted the request. For system clients, the resource management system shall always reject reservation requests from system contenders. A system client shall never directly preempt resources, or do so without the required negotiation acceptance. (Note that a system contender negotiating with another system client will fail, because system clients are assumed not to support negotiations.)

If a non-intrusive reservation fails, the contender may analyze why it failed. This can be done by processing the results of the reservation command returned by the Resource Manager. Any application can use `Fcm::GetReservationStatus` to learn about various reservation properties that hold for the resource. In general, applications should subscribe to `ReserveIndication` and `ReleaseIndication` events if they need to be aware of reservations and releases, including those related to resources they have reserved themselves.

For a group of resources, it can be specified whether a non-intrusive or preemption approach should be used in a reservation. However, only if all resources can be reserved, will they indeed be reserved. Otherwise, none will be reserved.

Due to other reservations, missing resources, or network partitions, a reservation command may fail for one or more resources. After invoking `ResourceManager::Reserve` and `ResourceManager::Negotiate`, a contender should always consult the status or negotiation record for each individual resource.

Application requirements. At any time during a reservation, a primary client can receive one or more negotiation messages from Resource Managers. A client shall accept and process them immediately. The client should respond within the specified timeout period. It should receive a follow-up message from a Resource Manager if the preemption request is withdrawn. It is not necessary that a current client releases the requested resources; this will be done by the resource management system. The client can learn about the actual preemption by subscribing to the `ReserveIndication` event.

Example. Below is a scenario that sketches how a user and a system client would interact with the resource management system (abstracted by RM) and the resource (R).

In this scenario, a + after a method denotes an accept response on the command. A + or - on a dashed arrow denotes an accept or reject response on the earlier command (without + or -). NIR stands for non-intrusive reservation, GRS stands for `GetReservationStatus` (an FCM method).

The scenario illustrates that a user client may be using a resource, possibly being unaware of, say, an Action Scheduler, which operates as a system client. An Action Scheduler shall always attempt a non-intrusive reservation (NIR) first. If this fails, it will typically negotiate for the resources it needs. The user client is informed of the scheduled action that was planned. In this case, the user chooses to accept a preemption of the resource, so that the scheduled action can succeed.

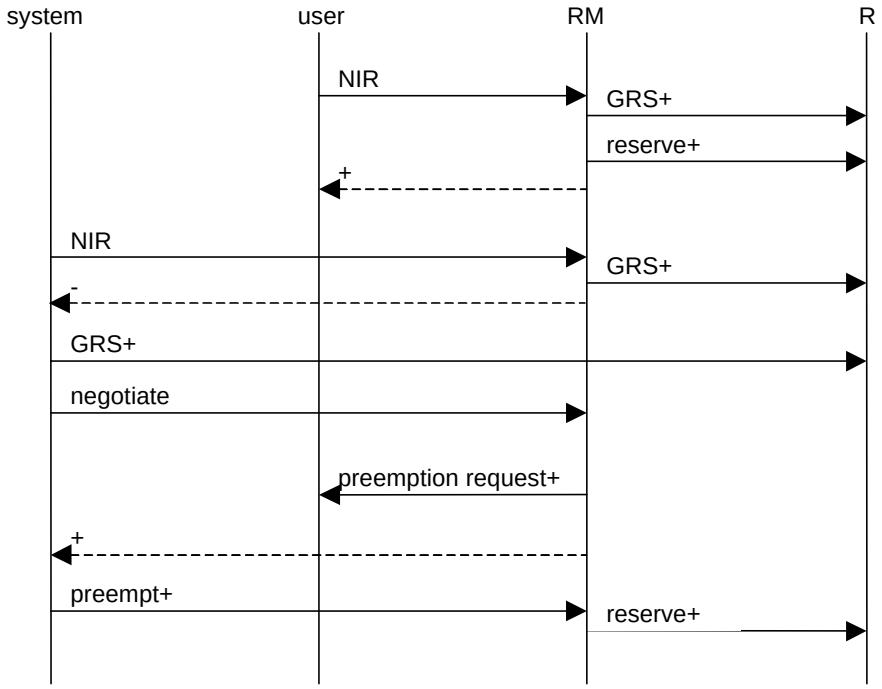

Figure 23. Reservation Protocol

3.8.4 Scheduled Action Management

The goal of Scheduled Action Management is to allow and secure Scheduled Action programming by the system. That is:

- to ensure that at *starting time*, all the necessary resources (FCM/DCM, bandwidth, …) are present and thus can be reserved

- to check at *scheduling time* the feasibility of the action at starting time i.e. to check the feasibility of the commands planned on the (device) resources and the planned inter-connections (network resources) between them

- to perform at *starting time* all the planned reservations and actions

Management of a Scheduled Action involves, apart from the Resource Manager, the following components:

- An application that specifies the scheduled action (the *invoking application*). The invoking application is only involved in the ordering of a Scheduled Action. If it disappears afterwards, the execution of the Scheduled Action is not affected.

- One or more FCMs or DCMs that are needed to successfully complete the required scheduled action.

- (Optionally) a *trigger* (application or FCM) that notifies the Resource Manager to start the action.

- (Optionally) an application that is involved in the scheduled action in the sense that it takes control over the involved resources during action time (*control application*).

Transactions (reservations, etc.) between FCMs, DCMs and applications such as described above are performed via a Resource Manager in an architectural model discussed below. First an overview of the data involved in a Scheduled Action will be given.

3.8.4.1 Scheduled Action Data

A Scheduled Action is defined by the following information:

- SEID of Scheduled Action controller (optional)
- SEID of Scheduled Action trigger (optional)
- Start Commands list (in a strict sequential order)
- Stop Commands list (in a strict sequential order)
- Connection list (in a strict sequential order)
- Start and Stop Time information
- Involved Resources HUID List
- User information (optional)

Scheduled Action controller: the application that will be awoken when the Scheduled Action is executed. This application may control the resources. If the field is absent, the Scheduled Action will not be controlled during execution but will execute autonomously.

Scheduled Action trigger: this field indicates the trigger (e.g., FCM or DCM) that will generate the actual notification for starting/stopping the execution of the Scheduled Action. The start/stop times will not be used in this case to actually start/stop the Scheduled Action's execution, but will be used to coordinate the reservation of resources with other schedules.

Start / Stop Commands: these are ordinary HAVi commands (defined by the APIs). They must be listed in order of execution and it has to be specified which resource must execute them. This sequential order is needed to ensure that a command cannot be sent to a resource without being sure

that the previous command has been successfully executed by the relevant resource. Commands can be either FCM commands or DCM commands.

Connection List: these indicate the network resource allocations needed for the Scheduled Action. From this list, the FCM plug connections that ultimately will be executed by the Stream Manager, will be constructed.

Start and StopTime Info: time information needed for scheduling an action (including date, start and stop time and periodicity data).

Involved Resources HUID List: this list represents the resources that are involved in the Scheduled Action.

User Information: this optional field can contain a description of the Scheduled Action that provides useful information in case a scheduled action that was originally accepted can no longer take place.

3.8.4.2 Scheduled Action Model

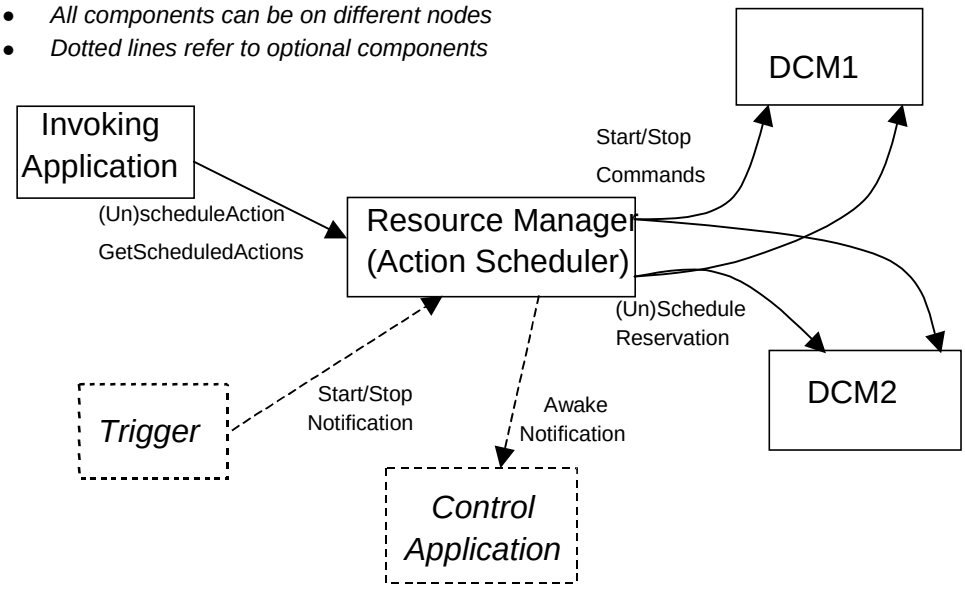

Figure 24. Resource Manager and Scheduled Actions

Figure 24 provides an overview of the involved software elements in a Scheduled Action. A description follows.

3.8.4.2.1 Scheduled Action

The invoking application passes its Scheduled Action (including Scheduled Action parameters as described above) to a Resource Manager of its own choice. The invoking application might choose its local Resource Manager, or take into account that the chosen Resource Manager can best be at the same node as one of the involved DCMs or trigger or control application (to decrease the chance of problems due to network changes). The Resource Manager chosen will be called *Action Scheduler* in the sequel.

The Action Scheduler persistently stores the Scheduled Action from the time the Scheduled Action

is invoked until the time it is completed and no longer needed. Per Scheduled Action only one Action Scheduler is involved. In case of network changes (e.g., network re-join after partition), it will take care that its Scheduled Actions are recovered (see below: Network Changes).

The Action Scheduler will assign a unique (local) index to the Scheduled Action for future reference. After acceptance of the Scheduled Action this index is eventually returned to the invoking application. A Resource Manager shall reuse these indexes as little as possible.

The invoking application is responsible for gathering all information about the involved FCM HUIDs, time info, connection list, etc. The model does not assume that all involved FCMs in a Scheduled Action are reservable FCMs: also non-reservable FCMs shall be accepted for Scheduled Actions.

Data Passed to Action Scheduler
SEID of control application(optional)
SEID of trigger (optional)
Operation code for awake notification message
Start Commands list
Stop Commands list
Connection list
Start time/date
Stop time/date
Periodicity
Involved FCMs HUIDs
User info (optional)

Below is the basic structure of a start or stop commands list (see also section 5.10.2). The format of the "command to be executed" is exactly the same as the (content of the) message that will take care of its execution. The Action Scheduler will not interpret the command itself (and considers it as a sequence of bytes).

Start or Stop Command List	
Command to be executed	HUID of the FCM or DCM that must execute it
...	...

The structure of the connection list allows the Action Scheduler to defer from it the eventual Stream Manager commands (FlowTo). The connection list looks as follows (see also API section):

Connection List		
Source FCM Plug	Sink FCM Plug	Stream Type
...	...	...

The invoking application might use a slightly different time for "start" then the one that was specified by the user, in order to take into account that the execution of all the actions can readily be done beforehand. The application can make use of `GetWorstCaseStartupTime` of the involved FCMs for calculating a worst case start-up time.

3.8.4.2.2 Schedule Reservation and DCM Checking

The Action Scheduler contacts the DCMs of each involved resource to distribute *Restricted Scheduled Actions*. Restricted Scheduled Actions are restricted versions of an original Scheduled Action that contain only the data that concern the resources local to each DCM. By this means, the FCMs local to the DCM can be checked (if the resource is free at scheduled time) and be aware of the actions (commands and connections) to perform at start or stop signal.

Data Passed to Involved DCMs
Start Commands list
Stop Commands list
Connection list
Start time/date
Stop time/date
Periodicity
Involved FCM HUIDs
User info (optional)

The start and stop command lists consist of a subset of the original Scheduled Action lists. Only those commands are gathered that belong to the particular involved DCM (the *target DCM*). The connection list is also constructed from the original Scheduled Action list. Only those entries are copied that refer to an FCM "inside" the target DCM. The same holds for the list of involved FCMs.

In this way each involved DCM is able to build up an internal agenda for reservations of resources. The internal agenda allows a DCM to check for schedule overlaps. For doing this the DCM has to take into account the start/stop time/date attributes and also the periodicity attribute. In case a DCM detects a schedule overlap, it checks whether it (and its FCMs) will be able to execute simultaneously the intended actions (start/stop commands and connections) during the (overlapping) time period.

A DCM shall at least check whether there is a schedule overlap. This can require checking of several schedules if periodicity was specified. Further checking of the involved actions is highly recommended because otherwise possible problems will only become apparent at starting time of the Scheduled Action.

Further notes on checking:

For connections from an internal FCM to an external FCM (internal/external with respect to the target DCM), only the DCM internal part has to be checked. The other parts are checked by another DCM and the bandwidth availability is checked separately (see section 3.8.4.2.3).

The Scheduled Action reservation of involved resources can lead to blocking of *dependent* resources. Dependent resources are separate resources (FCMs) which can no longer be used freely because of, e.g., already established connections. This means that different Scheduled Actions involving different FCMs (within the same DCM) can still be conflicting. The blocking of dependent resources is "invisible" to resource management: a DCM is responsible for taking into account the possibly resulting blocking of dependent resources when accepting a scheduled reservation.

A consequence of the above is that DCM checking can be rather complex. However, DCMs can also choose to do things rather simply by allowing only restricted reservations (e.g., every reservation reserves the complete DCM). The "checking complexity" that a DCM wants to deal with also determines how much information of the Restricted Scheduled Action the DCM has to store.

After checking, the DCM returns to the Action Scheduler whether it accepted (its part of) the Scheduled Action.

3.8.4.2.3 Bandwidth Checks

If all (local) DCM checks are OK, the Action Scheduler will check whether the needed network resources will be available at starting time. This check has to be performed only for connections between FCMs on different DCMs. The network resources to be checked for availability are bandwidth and channel numbers. Since these are global resources, the Action Scheduler that is responsible for a new schedule has to inquire all other Action Schedulers about already programmed scheduled actions that will overlap in time with the new schedule.

The checking, which shall be done at scheduling time, can be conceived as taking place in two steps:

1. The Action Scheduler collects all planned connections corresponding to existing schedules in the network that overlap in time with the new schedule. It uses the `ResourceManager::GetScheduledConnections` API for this purpose. In this way the Action Scheduler can build up a bandwidth table of needed bandwidth and channels during the period of the new schedule. (The actual bandwidth calculation method is described in section 5.10.5.)

2. The Action Scheduler then compares the network capacity (total available bandwidth and total available number of channels) with the information in the table and thus checks if there will be enough bandwidth (and channel numbers) at execution time of the new schedule.

Note that to perform the bandwidth related calculations the Action Scheduler may utilize already available Stream Manager facilities. Since all calls to the Stream Manager are local and the Action Scheduler and the Stream Manager will be from the same vendor, the Stream Manager services needed to do the calculations are proprietary and not part of this document.

3.8.4.2.4 Usage of Timers and Triggers

If the above bandwidth checking or one of the previous DCM checks failed, the Action Scheduler will cancel the Scheduled Action. The local scheduled reservations on the DCMs will be undone via `Dcm::UnscheduleReservation`.

If all checks are okay and no trigger is specified, the Action Scheduler will use a timer in order to get notified when the start/stop times happen. When no trigger is specified, the Action Scheduler will set the start time according to the next valid value of the timer.

The Action Scheduler may use a Clock FCM for setting up the timer start/stop times, however, the Action Scheduler should be able to complete the Scheduled Action (in a proprietary manner) also when no Clock FCM is available. For a timer facility used by the Action Scheduler (Clock FCM or proprietary) a worst case accuracy of one minute is required.

In case a trigger was specified, the possibly needed subscription is not dealt with by the Action Scheduler. The application is responsible for setting up the trigger notifications. If the trigger start notification is never sent, for one reason or the other, the Scheduled Action will remain existing (waiting for the notification). It is up to the user (application) to remove the Scheduled Action. The same holds if the trigger stop notification is never sent. The reservations done will remain until the Scheduled Action is removed.

3.8.4.2.5 Executing the Scheduled Action

When the Action Scheduler is notified (either by a timer or by the trigger) that the Scheduled Action should be executed, it first executes all reservations of the involved FCMs (via `ResourceManager::Reserve`). This is a non-intrusive reservation with client role of “system”

and requesting primary access rights. If the reservations fails the Action Scheduler will start negotiating as a cooperative controller (via `ResourceManager::Negotiate`, see section 3.8.3). Then it uses the connection list to establish `FlowTo` connections via the Stream Manager. Before establishing connections, the Action Scheduler turns on the power state of the DCMs, which are needed to establish the connection, to avoid connection failure. Each `FlowTo` should use "any" values for `ConnectionHint` except for stream type. Initially a `FlowTo` will be attempted with `dynamicBw` equal to `False`. If this fails, a `FlowTo` with `dynamicBw` equal to `True` will be attempted. After connections have been established, the Action Scheduler then executes all start commands. All these are HAVi commands of the format described in 3.2.3.4. The connections and commands are executed in the order initially given by the (invoking) application. In case the reservation fails, or a command or connection does not return `SUCCESS`, the Action Scheduler will abort the complete Scheduled Action and generate an `ErroneousScheduledAction` event.

In case a control application was initially specified, this application is awoken by the Action Scheduler via `<Client>::AwakeNotification`. Note that such application awaking is optional, and is not required to complete a Scheduled Action. In this case, the above reservations are not performed by the Action Scheduler but by the control application itself. This is done in order not to disturb the owner relationship of the resources. For the same reason the connections and command executions will be executed by the control application. This means that in case a control application was initially specified, the start/stop commands etc. of a Scheduled Action are only used for checking. This also means that the control application has to deal with failing reservations, etc.

If everything has been started, the Action Scheduler will use the specified method (timer or trigger) for stopping the Scheduled Action. When no trigger is specified, the Action Scheduler will set the stop time according to the next valid value of the timer.

3.8.4.2.6 Ending the Scheduled Action

When the Action Scheduler is notified (either by timer or trigger) that the Scheduled Action should be stopped, it executes all stop commands, breaks all connections (via `StreamManager::Drop`) and releases the resources (via `ResourceManager::Release`). Again, these are ordinary, existing HAVi commands. The commands are executed in the order initially given by the (invoking) application. If the Scheduled Action was not used with "periodicity", it will be removed from the list of Scheduled Actions in the Action Scheduler list and via `Dcm::UnscheduleReservation` also from the internal agendas of the involved DCMs. When the Scheduled Action is used with "periodicity" and no trigger is specified, the Action Scheduler will set the start time according to the next valid value of the timer. When the Scheduled Action is used with "periodicity" and a trigger is specified, the Action Scheduler will do nothing.

In case a control application was initially specified, the above actions are not executed: if the application did not already finish it will continue as in "normal" (unscheduled) operation. The removal of the Scheduled Action however will take place.

3.8.4.3 Query and Modification of Scheduled Actions

An application can consult the database of Scheduled Actions maintained by the Action Schedulers. It can use a facility to get an overview of all Scheduled Actions stored in a given Action Scheduler (`ResourceManager::GetLocalScheduledActions`), and can inspect individual Scheduled Actions in detail (`ResourceManager::GetScheduledActionData`).

Applications can use these facilities for removing no longer needed Scheduled Actions (e.g. periodic ones) via `ResourceManager::UnscheduleAction`. The Action Scheduler will then cancel the Scheduled Action and the scheduled reservations on the involved DCMs will be undone

via Dcm::UnscheduleReservation. A change in a Scheduled Action can only be performed by first removing it and then scheduling it again (with appropriate changes).

3.8.4.4 Network Changes

Here is the model showing how the Resource Manager takes into account changes in the HAVi network. After network change, all scheduled actions are rechecked. This may result in some warning messages, e.g., while changing the cabling.

The Action Scheduler keeps track ("watch-on") of all involved DCMs and (if specified) the trigger and control application. If one of the DCMs disappears due to a network change, the Action Scheduler will react as in case of an UnscheduleAction, although it will not remove its own copy of the (complete) Scheduled Action. Instead, it shall 'remember' that it is invalid. However, the Scheduled Action will not be executed as long as the missing resources are not back on the network. Note that the ResourceManager::UnscheduleAction involves Dcm::UnscheduleReservation, which might not be feasible for all the involved DCMs since some may have disappeared. The Action Scheduler shall generate an InvalidScheduledAction event with the user info of the Scheduled Action as parameter. In case the specified trigger or control application disappears, the AbortedScheduledAction event is generated (ErroneousScheduledAction if the Scheduled Action is executing) and the Scheduled Action is completely removed from the system (as by ResourceManager::UnscheduleAction). It is therefore recommended that a Resource Manager is chosen on the same node as at least the controller application.

- For each Scheduled Action, each DCM keeps track ("watch-on") of the associated Resource Manager (Action Scheduler). If the Action Scheduler disappears due to a network change, the DCM will remove the (Restricted) Scheduled Action from its internal agenda and generate an InvalidScheduledAction event with user info as parameter. Note that this can also happen when the Scheduled Action is already in execution.

- If a new DCM is installed on a node, the Action Scheduler is notified of the installation of the FCMs via NewSoftwareElement events, and it will check if an 'invalid' Scheduled Action is associated with them (by checking the HUIDs). If all resources are available again, the Action Scheduler will try to restart the Scheduled Action mechanism. If the restart fails (due to new Scheduled Actions in the mean time), an AbortedScheduledAction event is generated.

- After a network change the available bandwidth may decrease, and as a result some scheduled actions may no longer be able to proceed. Each Action Scheduler that has stored Scheduled Actions is responsible for rechecking the available bandwidth after a network reset event is received. An Action Scheduler that detects a lack of bandwidth generates an InvalidScheduledAction event to inform the user and makes the Scheduled Action invalid. If bandwidth becomes available again the Scheduled Action is re-installed.

- Even if the resources are still not available at start time, the Scheduled Action is not removed: it must be possible to launch the Scheduled Action even if it is delayed. Only if the Scheduled Action is still not feasible at stop time, and only if it is not a triggered Scheduled Action, the copy of the Scheduled Action in the Action Scheduler is removed and an event AbortedScheduledAction is generated to inform the user.

3.9 Application Modules

Application Modules are conceptually similar to DCMs. They provide an interface to a service that is usually provided by software only. An Application Module is a normal HAVi object in the sense that it communicates via the HAVi Messaging System. Like DCMs, Application Modules have HAVi Unique IDs. The HUID allows other applications to find the Application Module after partial system unavailability.

Since applications typically provide proprietary functionality, the standardized part of the API is small. It consists of basic identification (HUID) and visual representation (icon) and a means for providing a user interface. The user interface may be provided in two ways, corresponding to the two levels of interoperability: via DDI or via an uploadable havlet (similar to a havlet for a DCM). For an Application Module it is optional whether to support one or both ways of providing a user interface; what is supported is indicated by the `ATT_GUI_REQ` attribute in the Registry.

Application Modules can be provided by third-party application writers. Application Modules written in Java may be handled by arbitrary FAVs, they use the same format as DCM code units. The way IAVs handle Application Modules is proprietary. Also the way an FAV finds and installs third-party Application Modules (via a disk, Internet or any other means) is proprietary to the FAV node.

An FAV or IAV device is responsible for assigning the HUID to any Application Module running on the device. For Application Modules, two different HUID schemes provide different levels of persistency.

3.10 Code Unit Authentication

3.10.1 Outline of digital signature algorithm

All uploadable DCM code units shall be signed. Havlet code units may also be signed. HAVi specifies a public digital signature algorithm for these two kinds of code units. However, it may or may not be applied for Application Module code units, since verification for Application Module is vendor-dependent.

In HAVi Authentication, a public key system based on Rivest Shamir Adleman (RSA) is used. Also, Secure Hash Algorithm revision 1 (SHA-1) is used to digest the messages.

The HAVi Certification Authority (HCA) defines and keeps a unique pair of private key (never disclosed) and public key. The HAVi Certification Authority (HCA) also issues several pairs of private key and public key to each vendor on request, with the HCA's digital signature for the vendor unique public key. In this way, each vendor will have flexibility to be able to generate final signature to their products (DCM code units and havlet code units).

- In the HAVi specification of this version, HAVi public key is a 2048 bits key defined by HAVi, and statically stored in each FAV
- vendor unique public key is one of 1024, 1536 or 2048 bits key assigned by the HCA, and included in a certificate
- HAVi uses RSA algorithm for digital signature compliant to PKCS#1 v2.0 [19] (See section 3.10.1.1).
- HAVi uses well-known 160 bits SHA-1 Message Digest [20].

A HAVi-signed code unit includes some specific files necessary for authentication in the JAR file. When an FAV is to install a HAVi-signed code unit, it has to verify the code unit with these files in advance to install the software element.

Uploadable DCM code units and havlet code units may be located in different locations with different security risks. HAVi therefore distinguishes between "External code units" and "Embedded code units". "External code units" are code units originating from device external locations like a Web site or an unsecured packaged-medium, e.g. floppy. "Embedded code units" are code units located in the SDD of a BAV, in a DCM/Application module (for havlets), or in secure storage (e.g. resident storage like NV-RAM or HDD in FAV or BAV, or proprietary secured memory card).

Thus, each vendor can request to obtain one or more pairs of vendor-unique keys from the HCA. These keys may be different in its size or validity period. These keys may be used for only External code units, or for only Embedded code units, or both.

The following figure shows the outline structure of certificates/signatures.

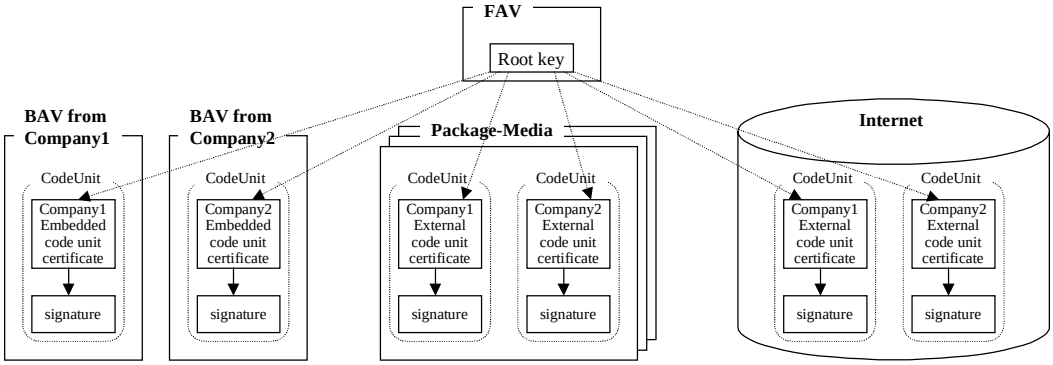

Figure 25. Certificate Tree

3.10.1.1 EMSA-PKCS1-v1_5 encoding method in HAVi

Since HAVi Authentication allows only SHA-1 hash function, the encoding method is as follows, which is compliant to EMSA-PKCS1-v1_5-Encode in PKCS#1 v2.0.

EMSA-HAVi(M, emLen)

Option: Hash SHA-1 hash function

Input: M Message to be encoded

emLen intended length in octets of the encoded message.

Output: EM encoded message, an octet string of length emLen.

Steps:

1. Apply the hash function to the message *M* to produce a hash value *H*:

H = *Hash*(*M*)2. Generate an octet string *PS* consisting of length *emLen* – 37 octets with value 0xFF.

3. Concatenate *PS* and *T* to form the encoded message *EM* as

$$EM = 01 \| PS \| 00 \| 3021300906052b0e03021a05000414 \| H$$

4. Output *EM*

3.10.2 Code Unit Format

A HAVi-compliant signed code unit shall contain three specific files, named ‘havi.hashfile’, ‘havi.signature’ and ‘havi.cert’ in the JAR file. If the JAR file has directories, then each directory may have its own ‘havi.hashfile’. Even in that case, the root directory shall have a ‘havi.hashfile’.

A HAVi-compliant signed code unit may contain another specific file, named ‘havi.crl’, if one or more vendor-unique public key(s) have ever been revoked.

Below is the structure of HAVi-compliant signed code unit (uploadable DCM, havlet and possibly Application Module).

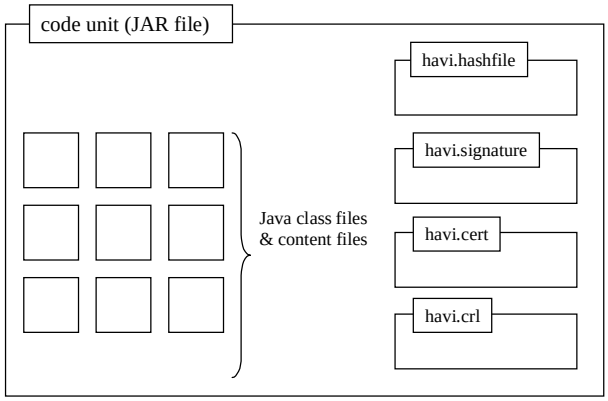

Figure 26. Authentication-specific files in a JAR file

3.10.2.1 Hash file

A code unit consists of several class files and possibly contents files. A file named ‘havi.hashfile’ is used to specify the filenames and their order to compute a SHA-1 Hash digest value for the directory.

The root directory shall have a master ‘havi.hashfile’. If class files and/or content files are stored in a subdirectory and need to be verified, such a subdirectory also contains its own ‘havi.hashfile’ in it.

The format of ‘havi.hashfile’ is compliant to ETSI TS 101 812 V1.1.1 (2000-07), section 12.4.1.1.

However, since HAVi only allows SHA-1 as the hash algorithm, the format is shrunk as follows:

```
Syntax                                    #bit Format

Hashfile () {
   digest_count  /* always 1 */                16 uimsbf
   for( i=0 ; i<digest_count ; i++ ) {
      digest_type  /* always 2 for SHA-1 */     8 uimsbf
      name_count                               16 uimsbf
      for( j=0 ; j<name_count ; j++ ) {
         name_length                            8 uimsbf
```

```
         for( k=0 ; k<name_length ; k++ ) {
            name_byte                              8 bslbf
         }
      }
      for( j=0 ; j<digest_length ; j++ ) {  /* always 20 for
SHA-1 */
         digest_byte                               8 bslbf
      }
   }
}
```

`digest_count` is set to 1 because all the files are digested by SHA-1.

`digest_type` is set to 2 to represent SHA-1.

`digest_length` is 20 (bytes) because SHA-1 generates 160 bits digest.

`name_count`: This value identifies the number of object names associated with the digest value. It may be one or more.

`name_length`: This value identifies the number of bytes in the object name

`name_byte`: This value holds one byte of the object name. Terminating null characters are not considered to be part of the file name.

The digest value is computed over the objects named in the havi.hashfile in the ordered list. The ordered list may contain an arbitrary mix of different object types (that is a mixture of file and directory names). The digest value is computed over the concatenated relevant data in the order. The relevant data for each objects depends on its type:

- if the object is a file, relevant data is the entire content of the file.
- if the object is a directory, relevant data is the content of the havi.hashfile of the named directory.

3.10.2.2 Signature File

The SHA-1 Hash digest value in the 'havi.hashfile' (a sequence of `digest_byte`) for the root-directory is signed and contained as a file named 'havi.signature'.

This file is generated by the vendor who creates the code unit, with one of the 'vendor-unique private key' issued by HAVi to the vendor.

The file format of 'havi.signature' is as follows:

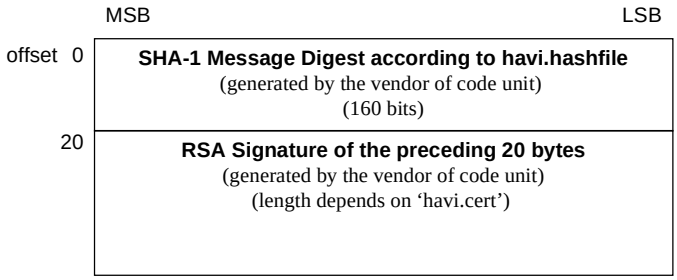

Figure 27. havi.signature format

3.10.2.3 Certificate File

The signature file 'havi.signature' is verified with the 'vendor-unique public key' which corresponds to the vendor-unique private key that is used to generate the signature.

The 'vendor-unique public key' must be certified by the HCA. A file named 'havi.cert' is used for this purpose. The verifier module in an FAV can find the 'vendor-unique public key' to verify the 'havi.signature' in this file, and shall verify whether the public key is trusted.

'havi.cert' and corresponding vendor-unique private key will be issued to vendors from the HCA.

The file format of 'havi.cert' is as follows.

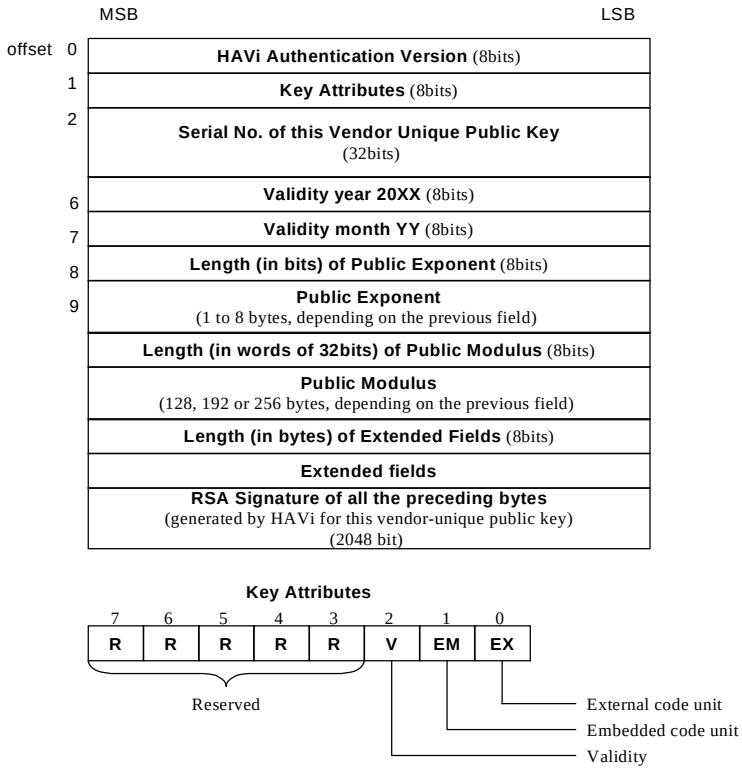

Figure 28. havi.cert format

`HAVi Authentication Version` – the version number of HAVi Authentication Algorithm. This field shall be set to 0x01 for HAVi Specification Version 1.1.

`Key Attributes` – each bit indicates the characteristic of this vendor-unique key.

`Serial No. of this vendor-unique public key` – a serial number uniquely assigned by HAVi for the key.

`Validity year` – year until when this key is valid. The last 2 digits of year (i.e. 00 – 99) is used and represented as a signed byte (i.e. ranging from 0x00 to 0x63).

`Validity month` – month until when this key is valid. 01 – 12 is used and represented as a signed byte (i.e. ranging from 0x01 to 0x0c).

For the Key Attributes field, each bit represents as follows:.

`Reserved bits` – not used in this version. These bits shall be set to 0.

`Validity bit` – 1 if the key has validity, 0 for otherwise. If this bit is set to 0, the verifier shall neglect the following Validity fields.

`Embedded code unit bit` – 1 if the key may be used for Embedded code units, 0 for otherwise.

`External code unit bit` – 1 if the key may be used for External code units, 0 for otherwise.

`Length of Public Exponent` – length of the following `Public Exponent` field, expressed in bits.

`Public Exponent` – a sequence of Public Exponent (in terms of PKCS#1 v2.0) of size `Length of Public Exponent`

`Length of Public Modulus` – length of the following `Public Modulus` field, expressed in words of 32 bits

`Public Modulus` – a sequence of Public Modulus (in terms of PKCS#1 v2.0) of size `Length of Public Modulus`

`Length of Extended Fields` – length of any extended fields which may be added in the future version. This field shall be 0x00 for the current version.

`Extended Fields` – This field may contain additional data for succeeding versions of the specification. In this version of the specification the field is absent, i.e., length of Extended Fields is zero.

`RSA Signature` – a sequence of 2048 bits RSA signature for all the preceding bytes, generated by the HCA for the key.

3.10.2.4 Certificate Revocation List File

A vendor-unique key may be revoked if the key is compromised. A HAVi-signed code unit may contain a list of revoked keys with HAVi's signature on it. Such a list is called Certificate

Revocation List (CRL), and has a fixed filename of ‘havi.crl’ in the root directory of the code unit JAR file. The CRL file will be issued by the HCA and delivered to each vendor when needed. The CRL may contain all the keys which have ever been revoked, or may contain only limited but significant keys.

Each DCM/havlet code unit vendor shall include the most recently issued ‘havi.crl’, if available, in the JAR file.

The file format of ‘havi.crl’ is as follows:

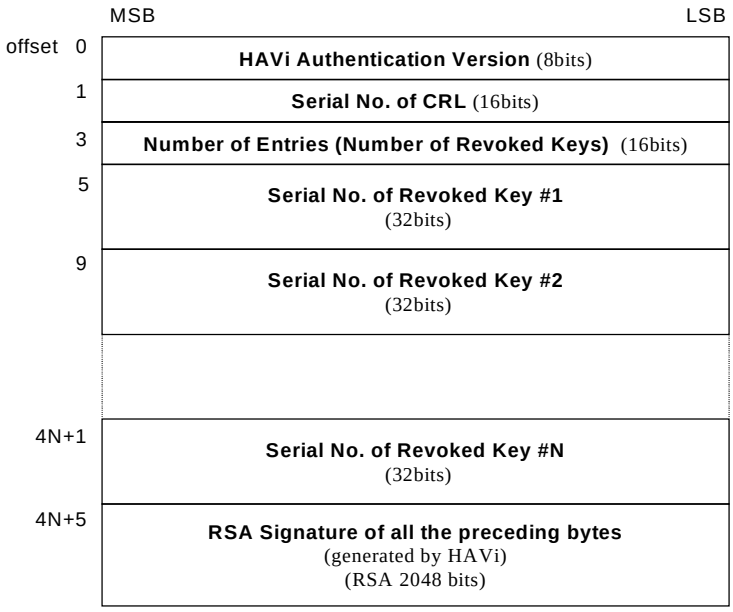

Figure 29. havi.crl format

`HAVi Authentication Version` – the version number of HAVi Authentication Algorithm. This field shall be set to 0x01 for HAVi Specification Version 1.1.

`Serial No. of CRL` – serial number of the CRL. A CRL of serial number = 0 does not exist, since 0 is reserved to mean that no keys have been revoked. When the HCA creates and delivers a new (updated) CRL, the CA will increment this value by one.

`Number of Entries` – indicates the number of `Serial No. of Revoked Key` s contained in the CRL

`Serial No. of Revoked Key` – serial number of each revoked key, which was assigned by the HCA and contained in corresponding ‘havi.cert’ file.

`RSA Signature` – a sequence of 2048 bits RSA signature for all the preceding bytes, generated by the HCA for the key.

3.10.2.5 Implementation note on keys, digest values and signatures encoding

Keys, signatures and digest values, when included in one of the four authentication specific files

(`havi.hashfile', `havi.signature', `havi.cert' and `havi.crl') shall always be seen as bit strings with leftmost bit first (bslbf format). If they need to be converted from (respectively to) integer values, primitive I2OSP (respectively OS2IP) described in PKCS#1 v2.0 shall be used. This corresponds to Big endian order.

PKCS#1 v2.0 explicitly describes all conversions that are needed regarding digest values and signatures. Key values (public exponent and public modulus) should be converted to integer values before being used in RSA algorithm.

3.10.3 Certificate Generation Procedure

When a DCM/Application Module code unit is to be signed, the necessary files are generated as follows:

- using SHA-1 Hash function, generate a 'havi.hashfile' file for the root directory of the JAR file, according to the procedure defined by ETSI TS 101 812 V1.1.1 (2000-07), section 13.4.1.1. If class files and/or content files are stored in a subdirectory and need to be verified, generation of 'havi.hashfile's for such subdirectories is needed before the 'havi.hashfile' file for the parent directory is generated.
- 'havi.cert' and corresponding vendor-unique private key will be issued to vendors by the HCA.
- using RSA algorithm, generate the digital signature for the SHA-1 Hash Digest value in the 'havi.hashfile' (a sequence of `digest_byte`) for the root-directory with the private key which is confidentially given to the vendor. The vendor should use a vendor-unique key of appropriate `Key Attributes`. Then generate a 'havi.signature' file which consists of the Hash Digest value and the signature.
- put the 'havi.hashfile' and 'havi.signature' described above, and the 'havi.cert' which corresponds to the vendor-unique key into the code unit JAR file. If one or more 'havi.crl' file(s) have ever issued by HAVi, put the latest 'havi.crl' into the JAR file.

3.10.4 Code Unit Authentication Procedure

When an FAV is to install a DCM/havlet code unit, the procedure is as follows:

3.10.4.1 DCM code unit install

- check if the code unit JAR file includes 'havi.cert', 'havi.signature' and 'havi.hashfile'. If any of these files are absent, the code unit is immediately regarded as "*untrusted*" and DCM installation fails. If all these three files are present, go to the next step.
- If the DCM manager has uploaded the code unit from the SDD of an BAV, check whether the `Embedded code unit` bit in the `Key Attributes` field is set to 1. Otherwise (the DCM Manager has downloaded the code unit from Internet), check whether the `External code unit` bit is set to 1. If it fails, the code unit is regarded as "*untrusted*" and DCM installation fails. If it succeeds, go to the next step.
- If `Validity bit` is set to 1, check whether the current date is within the validity period. If it fails, the code unit is regarded as "*untrusted*" and DCM installation fails. If it succeeds or `Validity bit` is set to 0, go to the next step.
- check whether the `Serial No. of this Vendor Unique Public Key` is identical to one of the revoked keys which the FAV maintains. If the key has been revoked, the code unit is regarded as "*untrusted*" and DCM installation fails. Otherwise, go to the next step.
- using the known (and stored in the FAV) HAVi public key, verify the 'havi.cert' (see section 3.10.4.3). If it fails, the code unit is regarded as "*untrusted*" and DCM installation fails. If it succeeds, go to the next step.
- check whether the hash values in 'havi.signature' and 'havi.hashfile' (a sequence of `digest_byte`) are the same. If it fails, the code unit is regarded as "*untrusted*" and DCM

installation fails. If it succeeds, go to the next step.
- using the vendor unique public key verified above, verify 'havi.signature'. If it fails, the code unit is regarded as "*untrusted*" and DCM installation fails. Otherwise, go to the next step..
- using SHA-1 Hash function, calculate a hash value according to the order specified by 'havi.hashfile'. Then compare the value with the value in the 'havi.hashfile' (a sequence of `digest_byte`). If it fails, the code unit is regarded as "*untrusted*" and DCM installation fails.
- if all these processes succeed, the DCM code unit is regarded as "*trusted*" and allowed to be installed.

3.10.4.2 havlet code unit install

- check if the code unit JAR file includes 'havi.cert', 'havi.signature' and 'havi.hashfile'. If any of these files are absent, the code unit is immediately regarded as "*untrusted*". If all these three files are present, go to the next step.
- If `Validity bit` is set to 1, check whether the current date is within the validity period. If it fails, the code unit is regarded as "*untrusted*". If it succeeds or `Validity bit` is set to 0, go to the next step.
- check whether the `Serial No. of this Vendor Unique Public Key` is identical to one of the revoked keys which the FAV maintains. If the key has been revoked, the code unit is regarded as "*untrusted*". Otherwise, go to the next step.
- using the known (and stored in the FAV) HAVi public key, verify the 'havi.cert' (see section 3.10.4.3). If it fails, the code unit is regarded as "*untrusted*". If it succeeds, go to the next step.
- using the vendor unique public key verified above, verify 'havi.signature'. If it fails, the code unit is regarded as "*untrusted*". Otherwise, go to the next step.
- check whether the hash values in 'havi.signature' and 'havi.hashfile' (a sequence of `digest_byte`) are the same. If it fails, the code unit is regarded as "*untrusted*". Otherwise, go to the next step.
- using SHA-1 Hash function, calculate a hash value according to the order specified by 'havi.hashfile'. Then compare the value with the value in the 'havi.hashfile' (a sequence of `digest_byte`). If it fails, the code unit is regarded as "*untrusted*".
- if all these processes succeed, the havlet code unit is regarded as "*trusted*".

For havlet code units, the FAV may load the "untrusted" code units into the Java runtime. However, even when loading such untrusted havlet code units is allowed, it is recommended that the FAV have a proprietary mechanism to assure that such an installation is only done with the user's responsibility.

3.10.4.3 Verifier Implementation Note

In a future version of HAVi specification, the authentication mechanism may be enhanced. In such a case, the later version will be updated so that it has backward-compatibility to this specification.

Therefore, a verifier implementation shall take it into account and comply the following rules:

- Even if the value of `HAVi Authentication Version` in 'havi.cert' or 'havi.crl' file is greater than 0x01, the verifier shall not regard it as a failure.
- If the value of `HAVi Authentication Version` in 'havi.cert' file is greater than 0x01, the value of `Length of Extended Fields` may not equal to 0x00. In such a case, some extended fields may be added before `RSA Signature` field. The verifier shall also verify the value of `Length of Extended Fields` and succeeding unknown sequence of bytes in verifying the signature of 'havi.cert' file. Of course the sequence of bytes does not make sense and thus will be neglected in the authentication process of this version.

3.10.5 Revocation

Each vendor is strictly responsible for keeping its own vendor unique private key confidential. In case a vendor unique private key has leaked out or compromised, some FAVs may refuse to regard a code unit which is signed by such private key as “trusted”. This is so-called “revocation” process. Revocation can be performed by detecting a vendor unique public key in a ‘havi.cert’ which corresponds to one of such revoked vendor unique private keys maintained in the FAV during authentication procedure. HAVi defines a standard Certificate Revocation List format (See section 3.10.2.4). Each FAV shall implement a Key Revocation Mechanism.

The procedure of the key revocation mechanism for DCM and havlet code units is as follows:

- an FAV maintains whole or a part of the latest CRL with `Serial No. of CRL` on a persistent storage inside the FAV. An FAV shall be capable of maintaining at least 100 `Serial No. of Revoked Key` entries.
- the FAV will update its CRL when and only when it finds a ‘havi.crl’ with greater `Serial No. of CRL` and valid CA signature on it.
- the FAV will neglect a ‘havi.crl’ with lower `Serial No. of CRL` than the one it keeps even if it has a valid CA signature.

The loading of an HAVi-signed code unit signed with a revoked key will fail in the authentication procedure. The HAVi-signed code unit will be seen as untrusted. Revocation of keys may lead to malfunctions. Overcoming that problem is easy in the case of External code unit (a new signature with a new key is to be computed) but rather difficult in case of Embedded code units. It is thus strongly recommended for the manufacturers to have different keys to sign External code units and Embedded code units.

Loaded HAVi-signed code unit signed with a revoked key are not required to be removed by the FAV. They may thus keep to be viewed as trusted.

It might also happen the HAVi CA key to get compromised. In that case, a new HAVi key may be issued. Each company may have to get a new certificate for its current key and to change the file `'havi.cert'` in all their available files. The old HAVi key shall be used neither for signing new code units nor to be embedded in new devices.

Each FAV shall have a mechanism allowing the user to turn off the security. When security will be turned off, all new downloaded code units will be seen as trusted. A FAV with an old security HAVi key will then still be able to run already loaded code units and to download new code units.

FAV may optionally have some proprietary and not publicly exposed mechanisms allowing to get the new HAVi Root key. These mechanisms shall strongly guarantee the origin of the new key. A FAV with a new Root key will behave as follows:

- All code units authenticated by the new HAVi Root key will be installed following the rules of section 3.4
- DCM code units authenticated by the old HAVi Root key will be installed following the rules of section 3.4.1 with the following restriction:
  - Embedded Code Unit bit in the Key Attributes shall be set to 1 and External Code Unit shall be set to 0.
  - An FAV with such a mechanism shall be able to store at least 2 old root keys.

3.10.6 HAVi certification procedures

The procedures related to the HAVi Certification Authority, e.g. obtaining keys and certificates, are described in [16].

4 Data Driven Interaction

A HAVi software element may provide a user with the ability to control another software element using the HAVi Data Driven Interaction mechanism. Within this interaction the first software element is termed the *DDI Controller* and the second software element the *DDI Target*. The DDI Controller uses a description of the UI to be presented to the user, *DDI data* (consisting of a set of *DDI elements*), obtained from the DDI Target. The means by which this control is accomplished is described in general terms below. Section 5.12 *APIs for Data Driven Interaction* contains detailed information on DDI-related data types and operations.

4.1 Data Driven Interaction Protocol

A DDI Controller and its DDI Target are both HAVi software elements each executing on an IAV or FAV device. The software elements may be on the same or different devices, implemented using native code or Java bytecode; in all cases, though, they interact by sending HAVi messages to each other. The DDI Controller communicates with the user by using the input and output devices of (typically) the device upon which the controller is executing. This I/O communication may be done in a DDI Controller implementation-dependent manner. The DDI Target may be a DCM that controls its device in an implementation-dependent manner. The HAVi DDI protocol described in this section is structured so that a DDI Controller may be written in a generic manner. That is, a DDI Controller need *not* be implemented with knowledge of a particular DDI Target in mind; *all* target-dependencies are represented in the DDI data (below) provided by the target to the controller.

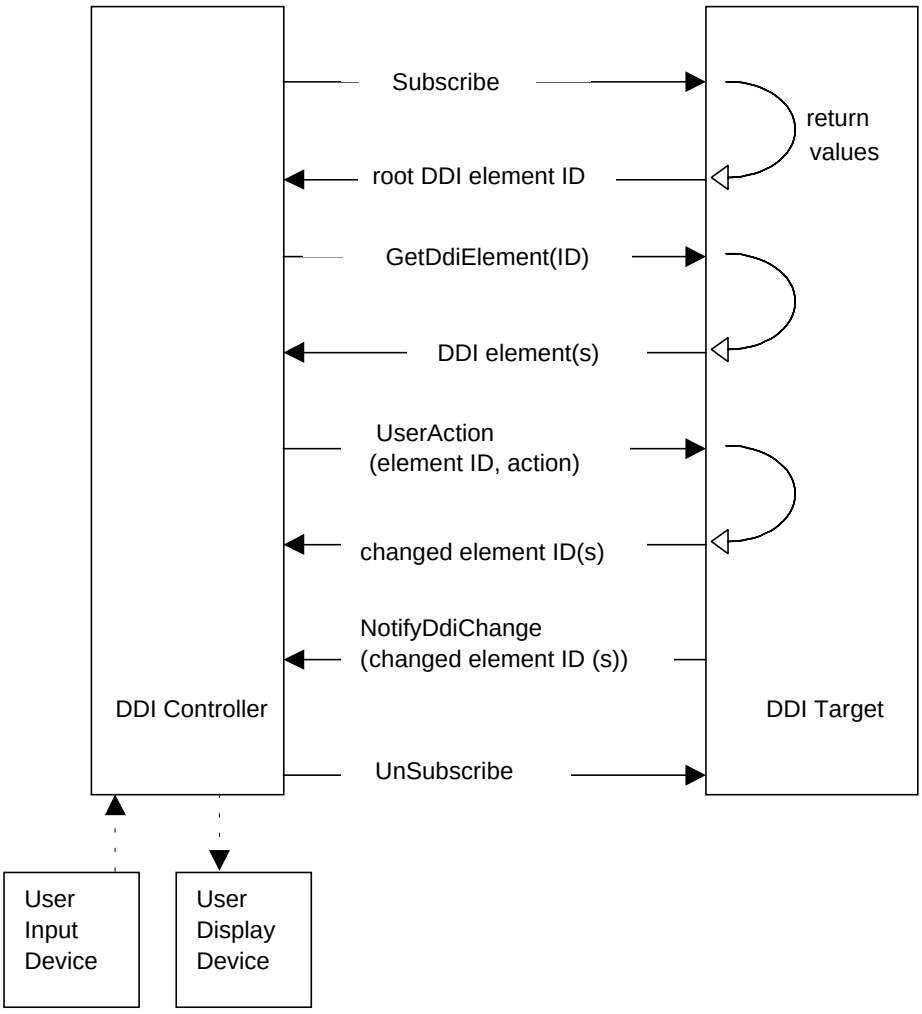

Figure 30. DDI Message Sequence Scheme (Typical)

As shown in Figure 30, controller-target interaction starts with the controller sending a `Subscribe` message to the target. The target will remember which software element sent this message (for use as the destination software element in possible `NotifyDdiChange` messages below) and return the ID of an initial (or *root*) DDI element (see below). The controller will use the `GetDdiElement` operation with this ID as an argument (or other DDI defined operations) to obtain the complete contents of the root DDI element for rendering on the controller's display device. Thereafter, based on user input, the contents of the controller's display device, and the DDI element(s) the controller has obtained from the target the controller may:

- change the information that is being displayed to the user using a controller implementation-dependent mechanism
- ask the target for another DDI element using the `GetDdiElement` operation (with the ID of the desired DDI element as an argument)
- or send a control command (determined by the contents of the DDI element and possible user input) to the target using the `UserAction` operation.

At any time after a controller has subscribed to a target and before that controller has unsubscribed to that target the target may indicate that the DDI description held by the controller has changed (indicating, for example, that some aspect of the target's internal state which is relevant to the user

has changed). The indication will be done by the target sending a `NotifyDdiChange` message to the controller. Arguments to this message will provide the controller with information about which parts of the target's DDI description have changed.

A `UserAction` message sent to the target by the controller will return to the controller information about those parts of the target's DDI description that have changed as a direct result of the action. For example, a DDI button hit may attempt to place a target device into a "rewinding" mode and the target may wish to indicate the success (or failure) of the command by returning a text element with an appropriate string value.

In this manner the user and the target device communicate with each other using a sequence of messages sent between the controller and the target guided by the DDI data the controller obtains from the target.

It is controller's responsibility to provide "exit" possibilities to the user in an appropriate way. When the controller is done with the interaction with a target it will send an `Unsubscribe` message to that target. The target will thereafter not send `NotifyDdiChange` messages to that controller and the controller should not send any other messages to that target. The controller may open a new interaction by sending another `Subscribe` message to that target.

In addition to the `GetDdiElement` operation (which, as briefly described above, takes as an argument the ID of a single element and returns the actual element with that ID) a DDI Target provides a number of other similar operations for returning *lists* of element IDs or lists of actual elements. The values of certain "large" attributes (e.g., bitmaps) are obtained by the controller from the target using the `GetDdiContent` operation. See section 5.12 *APIs for Data Driven Interaction* for details.

Note, because the HAVi DDI mechanism is primarily intended to allow users to interact with devices, the DDI Controller that "pulls" the DDI description functions essentially as a UI-controller and the target that supplies the DDI description is typically a DCM. However, it is possible for any application (not only a UI-controller) to act as a DDI Controller and for any application (not only a DCM) to act as a DDI Target. It is also possible for a DDI Target to be controlled by more than one DDI Controller and for a DDI Controller to control more than one DDI Target. The DDI data used by the DDI Controller need not have come from the DDI Target being controlled; though, typically it will come from the target. Any DDI data that is supplied by a DDI Target will always be appropriate for controlling that target.

Also note, for dealing with situations in which either the target or the controller goes away before an explicit `Unsubscribe` operation is sent, the Messaging System `MsgWatchOn` and `MsgWatchOff` calls should be used to notify each respective object that the other has gone away.

4.2 User Output and Input Device Models

The DDI Controller manages in an implementation-dependent manner a display device to present information to the user and a user input device to accept commands from the user. In order to allow the DDI model to apply to a broad range of particular devices these devices are modeled abstractly by only specifying in a general manner the way in which DDI data is associated with physical user interaction. The DDI data and its constituent DDI elements provide "suggestions" on how the controller is to make this association. These suggestions depend on the type and attribute values of the DDI elements.

4.2.1 Output Device Model

The DDI elements that the controller obtains from the target may be physically presented, *rendered*, to the user using the display device. Each DDI element is of a particular type (e.g., panel, icon, button) and each type of DDI element has a particular set of attributes (e.g., size, position, color, image content, sound content). All attributes are divided into two distinct classes: mandatory attributes and optional attributes. In a target's DDI data a DDI element's mandatory attributes will always have an associated value; optional attributes may or may not have an associated value. Every DDI element type has one or more mandatory *label* attributes whose values are text strings.

There are then three broad styles in which a DDI Controller can present a panel to a user:

Full-capability – all of the panel's elements are displayed as required by the element's attributes.

Intermediate – the entire panel cannot be displayed as given.

Basic – only a few or even a single element can be displayed at a time; however, at least, text string label attributes must be rendered.

See section 5.12.2.1 for recommended guidelines for non-full-capability controllers .

Note that the controller is also responsible for physically presenting the values of audio attributes of DDI elements as best as it and its display device are able.

Many DDI element types have attributes that are to be used by the controller to determine position and size on the display device. The physical display device is considered to be a rectangular array of discrete pixels. Position and size information is expressed with respect to a two-dimensional coordinate system for this rectangle with non-negative x and y coordinate values; the upper-left corner as the user faces the device is <0,0>. For details of coordinate system extent, aspect ratio of the mapping to physical pixels, etc., see the description of DDI panels in section *5.12.8 Individual DDI Elements*. The positions of DDI elements contained within *organizational* DDI elements (that is, panel and group elements) are relative to the position of the most immediately containing organizational element. The values that pixels may have and the physical interpretation of the value (e.g., color) are defined in section 5.12.4 *Basic DDI Types*.

For many types of DDI elements, attributes that specify their position are optional. When a position attribute is not given for an element the controller has broad freedom to locate the rendering of that element subject only to the guidelines provided by the place of that element within the overall DDI data – see section 4.3.1 *Organizational DDI Elements*.

4.2.2 Input Device Model

Like output to the user, input from the user is modeled abstractly. This model can be presented with the aid of the following definitions. DDI element types are *interactive* (e.g., button, set range) or *non-interactive* (e.g., status), based upon whether they, respectively, can or cannot be used by the controller to send a `UserAction` message to the target. DDI element types are *user-modifiable* (e.g. set range) or *non-user-modifiable* (e.g., button, icon) depending upon whether they, respectively, have or do not have a user-modifiable attribute.

User input is taken into account within the DDI model by requiring that a controller, in a manner appropriate for the actual physical input device, allow the user to:

- change the value of a user-modifiable attribute of a user-modifiable DDI element (e.g., enter a new text string into a DDI entry element) – this causes the controller to send a `UserAction` message to the target with arguments whose values depend on the type of the DDI element and contain the value supplied by the user.

- select an interactive DDI element (e.g., "hit" a DDI button element) – this causes the controller to send a `UserAction` message to the target with arguments whose values depend on the type of the DDI element and on the particular kind of selection the user performed.

- explicitly associate with the display device another DDI panel element by selecting a DDI panel link element. This will cause the controller to send a `UserAction` message to the target to indicate the selection. It will typically also cause the controller to obtain the DDI elements contained in this new panel from the target.

- change the display device by having the controller render DDI elements that are associated with the current DDI panel element but which are not currently rendered. This change only affects the display device; it does not directly cause the controller to send a `UserAction` message to the target. Note that if display resources are otherwise inadequate to render the new DDI elements, the controller may "un-render" DDI elements in a manner of its own choosing. For example, the controller may remove an arbitrary element from the current display or allow the user to control a scrolling mechanism.

4.3 DDI Elements

4.3.1 Organizational DDI Elements

The DDI elements contained in a target's DDI data are arranged into a hierarchy.

This hierarchy serves three purposes:

- It allows a controller to navigate through the DDI elements in an "organized" way.

- From the target's point of view, it indicates which DDI elements belong logically together and should, therefore, preferably be displayed physically together to the user.

- From the controller's point of view, it can be used to let the target know about which DDI element changes the controller should be notified.

Two types of *organizational* DDI elements determine the hierarchical organization of the elements in a target's DDI data:

- the DDI *panel* element, which has a (mandatory) attribute whose value is a list of the IDs of the DDI elements that are contained in the panel. DDI panel elements may *not* be (directly) contained in other DDI panel elements.

- the DDI *group* element, which has a (mandatory) attribute whose value is a list of the IDs of the DDI elements that are contained in the group. DDI panel elements may *not* be (directly) contained in other DDI group elements; however may contain other groups.

Panel and group elements are non-interactive and non-user-modifiable in the sense defined in section 4.2.2 *Input Device Model*.

A *non-organizational* DDI element is simply an element that is not an organizational DDI element. Non-organizational DDI elements will be discussed in section 4.3.3 *Non-Organizational DDI Elements*.

At any point in time a controller has a *current panel* which is the DDI panel element that the controller has most recently obtained from the target and which the controller is currently rendering as best it can (with all its contained DDI elements) on the display device.

The controller allows the user to navigate through the current panel's contents by means of the above hierarchy. The controller can choose to render panels, groups within panels and non-organizational DDI elements within panel or groups. The way in which a controller navigates through the panels and groups is its choice. The only thing that a target can assume is that the user is aware of the panel and the group that contains any DDI element the user may select. The controller (depending on the capabilities of its display device) can assure this in a number of ways:

- by displaying the complete panel, all its groups with all their non-organizational DDI elements
- by displaying for the panels and groups only their label or icon
- by displaying the non-organizational DDI elements only (in this case, the controller can still assure that the user knows what he or she is looking at, because the user has navigated through the panels, groups, and DDI elements in a way known to the user)
- some combination of the above.

4.3.2 Uses of Organizational DDI Elements

A DDI panel element (and its contained elements) may be used for the presentation and control of a function or a very closely related set of functions in the target device. The panel represents a set of DDI elements which the controller should render together on a single display screen. If this is not possible, the controller may divide/modify the set of panel elements over as few display screens as display capability allows. However, this should be done in a manner consistent with the intention of the designer of the target DDI data so that the user thinks of this set of elements as comprising a whole.

Similarly, a DDI group element may be used for the presentation and control of a sub-function of the target device. The elements contained in the group all have the same level of display priority. In situations where the controller cannot render all the groups and other elements in a panel at the exact positions specified by their attributes the controller may move or choose to (temporarily) not render some groups or other elements. However, the controller must make a strong attempt to keep the elements in a group together when they are rendered. The groups within a panel have a linear ordering. Like the panel and group elements which may contain them, non-organizational DDI elements have position information supplied by their attributes. If display resources are limited, the controller may change the position of non-organizational DDI elements within their panel or group.

4.3.3 Non-Organizational DDI Elements

As defined in the section above, non-organizational DDI elements are those DDI elements that are not panel or group elements. These types of DDI elements occupy "leaf" device locations in the DDI hierarchy determined by panel and group elements. A non-organizational DDI element has a type and type-specific mandatory and optional attributes that suggest to the controller:

- how the element should be rendered,
- what sort of effect user input should have upon the element; i.e., is the element *user-modifiable* in the sense defined in section 4.2.2 *Input Device Model*,
- and what effect user selection should have with respect to the controller sending `UserAction` messages to the target; i.e., is the element *interactive* in the sense defined in section 4.2.2 *Input Device Model*.

The text below briefly describes a subset of non-organizational DDI elements mentioning important mandatory attributes, indicating element use and typical renderings, and categorizing elements with respect to interactivity and user-modifiability. Detailed information is provided in section 5.12 *APIs for Data Driven Interaction*.

Text element – has a mandatory attribute containing a text string. This element is used to present a static label or other textual information to the user. A text element is non-user-modifiable.

Panel link element – has a mandatory attribute containing the ID of a panel element. Used for user-driven navigation by the controller. See section 4.4 *Navigation of the DDI Hierarchy*. A panel link element is non-user-modifiable.

Button element – has mandatory attributes that describe a sequence of "pressed" and "released" appearances. Used to allow a user to send a simple (i.e., without parameters) command to the controller. A button element is non-user-modifiable.

Choice element – has mandatory attributes that describe a discrete set of possible values of which the user may choose one or a number and thereby indicate to the target some command or course of action. Typical renderings are as many-out-of-many "choice boxes" or as one-out-of-many "radio buttons". A choice element is user-modifiable.

Entry element – allows a user to enter, for example, a text string value and send it to the target. A typical rendering is as a text entry field perhaps with an associated on-screen keyboard. An entry element is interactive and user-modifiable. Other numeric, date, and time data types may be also entered.

Animation element – has a mandatory attribute containing a sequence of icons. If there is only one icon in the sequence that icon is statically rendered by the controller. If there is more than one icon in the sequence the controller renders each icon in temporal sequence giving the user the effect of an animation. An animation element is non-user-modifiable.

Show Range element – has mandatory attributes defining a numeric range and a particular value within that range. Used to present static numeric information to the user. Typical renderings are as a circular meter with variable position pointer or as a linear variable length bar. A show range element is non-user-modifiable.

Set Range element – has mandatory attributes defining a numeric range. Used to allow a user to send a command with a numeric parameter to the target. Typical renderings are as a slider or dial. A set range element is user-modifiable.

All above mentioned DDI elements can be used in an interactive and non-interactive way. Again, the actual appearance and position of each DDI element depends on the controller; while a target's DDI data provides an explicit logical structure for its DDI elements, it only gives suggestions to the controller for their rendering.

4.4 Navigation of the DDI Hierarchy

4.4.1 Controller-Driven Navigation

If a controller chooses not to render a whole panel, then the controller must provide some means consistent with the capabilities of the user input and display output devices to allow the user to bring un-rendered elements into view. To allow the user to control this process, the controller may render items of its own choosing (e.g., arrows, scroll bars, etc.). These items are not obtained from the target and are specific to the controller implementation. However the controller implements controller-driven navigation, it must do so locally without explicit target involvement. That is, the controller may not during this process send `UserAction` messages to the target, though it (the controller) may obtain additional DDI elements from the target. This process is called *controller-driven navigation*. DDI elements may contain attributes which the controller may or may not use to guide the user during controller-driven navigation.

4.4.2 User-Driven Navigation

It is possible for DDI data to contain non-organizational DDI panel link elements. A panel link element has as the value of a mandatory attribute the ID of a panel element. Other attributes suggest a position and appearance for the rendering of the panel link element. This type of element offers a means for the controller to switch from one panel to another. If the user selects this element the controller may abandon rendering the currently rendered panel, make the specified panel element the current panel, and render it appropriately. This process is called *user-driven* navigation. Again, as for controller-driven navigation, the target will only be consulted if the controller needs additional DDI elements

DDI panel link elements thus make it possible for a target to specify DDI hierarchies of effectively arbitrary depth below the root panel. Also, there is no topological restriction for targets on the relationship between panels given by panel link elements; e.g., cycles are allowed at the discretion of the target designer.

4.5 Notification Scope for Target DDI Changes

As mentioned above in section 4.1 *Data Driven Interaction Protocol*, at any time while a controller is subscribed to a target, the target may indicate that its DDI description (some or all of which the controller may have previously read from the target) has changed by sending a `NotifyDdiChange` message to the controller. Arguments to this message will provide the controller with information about which parts of the target's DDI description have changed. The target may also indicate that its DDI description has changed in its response to a `UserAction` message sent to it by the controller. For large target DDI descriptions this may lead to many possibly extraneous notification messages being sent to the controller. A mechanism exists within the DDI model to reduce this message traffic.

It is possible for the controller to give the target a description of a portion of the target's DDI description, the current *notification scope*. The target will only notify the controller of changes to those target DDI elements that are within the notification scope. Target DDI elements outside of the current notification scope may change but the target will *not* send a corresponding indication to the controller. This indication is called a *change report* and may refer to zero or more changed DDI elements. A change report is included in the arguments for both the `UserAction` and `NotifyDdiChange` operations.

Any non-organizational target DDI element that changes and is within the current notification

scope will be included in the change report sent by the target to the controller. If DDI elements are added to or removed from organizational target DDI elements then these organizational elements will be included in the change report sent by the target to the controller. If both non-organizational and organizational target DDI elements are changed, both changes will be included.

For low notification-traffic situations, the controller can choose to set the notification scope to be the complete target DDI description. In higher traffic situations, the controller can set the notification scope to be the current panel. The controller may also decide to add each retrieved panel in its notification scope. A controller may change the notification scope during subscription.

Details are in section 5.12 *APIs for Data Driven Interaction.*

5 Software Element APIs and Protocols

5.1 HAVi Type Definitions and API Categories

5.1.1 HAVi API Descriptions

The following sections describe the APIs supported by the HAVi software elements. Each section begins with a summary *Services Provided* table which uses the terminology listed below:

- *Service*: The first column of a *Services Provided* table gives the service name. These names have three forms: *ApiName::Name*, *<Client>::Name* and *Name*. The first form indicates that the service uses a HAVi operation code from Annex 11.6. The second form indicates that the service would be provided by a client and uses an operation code selected by the client. The third form is used for events, procedure calls and callbacks.

- *Communication Type*: The form of communication used between the software element supporting the API (the "server") and a client, possibilities are:
  - *procedure call* (PC) – communication is via a local procedure call or via a HAVi defined Java API. This form of communication is initiated by the client. Implementation is platform dependent.
  - *call back* (CB) – communication is via a local call back function. This form of communication is initiated by the server. Implementation is platform dependent.
  - *messaging* (M) – communication is via the Messaging System. This form of communication is initiated by the client. The client sends a message containing an operation code listed in Annex 11.6 to the server. The server then sends a reply message.
  - *message back* (MB) – communication is via the Messaging System. This form of communication is initiated by the server. The server sends the client a message using an operation code previously provided by the client. The client may then send a reply message to the server. It is the responsibility of the client to select an operation code that does not conflict with those used in other messages it intends to receive.
  - *event* (E) – communication is via the Event Manager. This form of communication is initiated by the server. The server posts an event (sends a message to the Event Manager) which is delivered by the Event Manager using the event notification mechanism.

- *Locality*: Applies to communication via messaging (M), events (E) and message backs (MB). In the case of messaging, indicates whether an interface can only be called from software elements on the same device (local) or also from software elements on other devices (global). In the case of events, indicates whether the event is posted locally or globally (see `EventManager::PostEvent`). In the case of message backs, indicates whether the message can be sent only to software elements on the same device or also to software elements on other devices. Note – communication via procedure call (PC) and call back (CB) is always between local software elements.

- *Access*: Applies to communication via messaging (M), procedure call (PC), events (E), and message backs (MB). For messaging and procedure call, this entry indicates the software element types that may call an interface without generating an access violation. For events two values are given for this entry, the first indicates which software elements may post the event without generating an access violation, the second, shown in parentheses, indicates possible receivers of the event (this value is informative only). Similarly for message backs two values are given – the first indicates the software element which may send the message, the second value, shown in parentheses, indicates possible receivers of the message. Possible values for *Access* are:
  - *system* – can only be called from HAVi defined software elements. Specific software elements may be named (e.g., "DCM Manager")
  - *trusted* – can only be called from trusted software elements
  - *all* – can be called from any software element.
- *Reservation Protection*: Applies to DCM and all FCM interfaces (for the generic FCM APIs as well as for the type-specific FCM APIs). For DCM interfaces this refers to interfaces in which reserved FCMs are involved. Reservation protection will only be indicated for the above mentioned interfaces. In case an FCM is reserved, it indicates whether an interface will check that the (SEID of the) calling software element is identical to the (SEID of the) reserver and will refuse to execute a command that leads to reservation violation. If the FCM is not reserved, any interface call by any software element is allowed. In general, all state changing APIs should have a reservation protection indication. However, state changing APIs that do not compromise the functionality offered to a (primary or secondary) reserver are allowed to omit the reservation protection indication.

Entries in the *Services Provided* table for which there is no choice (e.g., *Locality* in the case of PC communication) or not applicable (e.g., *Access* in the case of CB communication) are left blank.

5.1.2 Basic HAVi Types

uint64

An unsigned 64 bit integer:

```
typedef unsigned long long uint64;
```

uint

An unsigned 32 bit integer:

```
typedef unsigned long uint;
```

ushort

An unsigned 16 bit integer:

```
typedef unsigned short ushort;
```

uchar

An unsigned 8 bit integer:

```
typedef unsigned char uchar;
```

GUID

```
typedef octet GUID[8];
```

VendorId

```
typedef octet VendorId[3];
```

The `VendorId` is the 24-bit SDD device vendor identifier, this is also the first three bytes of the GUID.

SEID

```
typedef octet SEID[10];
```

ApiCode

```
typedef ushort ApiCode;
```

Each group of APIs (such the Registry APIs or the Event Manager APIs etc.) is assigned a unique 16-bit `ApiCode`, these are listed in Annex 11.5. Note that a software element may accept messages from several API groups.

OperationCode

Messages sent to software elements contain a 24-bit operation code with the following structure:

```
struct OperationCode {
   ApiCode apiCode;
   uchar   operationId;
};
```

The `OperationCode` values for HAVi messages are listed in Annex 11.6.

Status

HAVi APIs generally return the following status structure:

```
struct Status {
   ApiCode apiCode;
   ushort  errCode;
};
```

The possible `Status` values are listed in Annex 11.7.

Version

```
typedef uint Version;
```

All HAVi software elements have an associated version number.

MediaFormatId

```
struct MediaFormatId {
   VendorId   vendorId;
   uchar      category;
   ushort     majorType;
```

```
   ushort     minorType;
};
```

AV recording and playback media, such as CD discs or DV tapes, are assigned a media format code. Possible values are listed in Annex 11.10.

ImageTypeId

```
struct ImageTypeId {
   VendorId   vendorId;
   ushort     typeNo;
};
```

Values of `ImageTypeId` specified by HAVi are listed in Annex 11.13.

StreamTypeId

```
struct StreamTypeId {
   VendorId   vendorId;
   ushort     typeNo;
};
```

Values of `StreamTypeId` specified by HAVi are listed in Annex 11.11.

CompOperation

Definition

```
typedef ushort CompOperation;
```

Description

This type is used for Registry queries and FCM notifications, it indicates the comparison operation to be performed on some attribute. The comparison operations supported by HAVi are listed in Annex 11.17.

Two families of comparison operations are provided:

- byte row comparisons – these operations do not interpret any of the bytes in a specified row (attribute value). For the basic IDL types (`boolean`, `char`, `octet`, `short`, `long`, `long long`) this corresponds with the natural comparison on these types.

- sequence comparisons – these operations interpret a row of bytes (attribute value) as a sequence of some type. In particular, the first four bytes of the attribute value are interpreted as the length of the sequence. The comparison is then done over the rest of the attribute value as a row of bytes. If the attribute value cannot be interpreted properly as a sequence (it does not contain four bytes), the comparison fails. For the sequences of basic IDL types, e.g., strings, this corresponds with the natural comparison on these types.

The most significant bit of a comparison operator indicates whether it is a byte row comparison or a sequence comparison.

DateTime

```
struct DateTime {
   ushort  year;
```

```
    uchar    month;
    uchar    day;
    uchar    dayOfWeek;
    uchar    hour;
    uchar    minute;
    uchar    sec;
    ushort   msec;
};
```

The possible values of the `DateTime` fields are listed below:

Field	Description
`year`	Year AD (1, …, 32767), ignored when value is 0xffff
`month`	Month (1,...,12), ignored when value is 0xff
`day`	Day of month (1, ..., 31), ignored when value is 0xff
`dayOfWeek`	Sunday, Monday,… (1,…,7), ignored when value is 0xff
`hour`	Hour of day (0, ..., 23), ignored when value is 0xff
`minute`	Minutes of hour (0, ..., 59), ignored when value is 0xff
`sec`	Seconds of minute (0, ..., 59), ignored when value is 0xff
`msec`	Milliseconds of second (0, …, 999), ignored when value is 0xffff

In addition to the rules for the range of valid values specified above, the following rules must also hold:

- `year`, `month`, and `day` values should be consistent with valid calendar values, unless one or more of them are set to "ignored". For example, February 30 is invalid, and February 29 is only valid in a leap year.

- The value of `dayOfWeek` should be consistent with the `year`, `month`, and `day` values provided, unless one or more of them are set to "ignored".

If an invalid `DateTime` value is specified as an in parameter to an API call, an `EINVALID_PARAM` error code may be returned.

5.1.3 Error Handling

All HAVi APIs involving messaging (those APIs where the *Communication Type* is "M" or "MB") shall return the `Status` structure specified above. The `Status` structure consists of two fields: an API code and an error code. Generally the different software elements will define their own error codes (see Annex 11.7), in addition though there are several "general purpose" error codes that can be used by any software element. These general error codes are:

- `SUCCESS` – the operation has succeeded (this is the normal return value in `Status` and not an error)
- `EUNKNOWN_MESSAGE` – the receiver of a HAVi message does not support the API indicated by the Operation Code contained within the message
- `EACCESS_VIOLATION` – the caller of an API does not have permission to perform the operation
- `EUNIDENTIFIED_FAILURE` – an error of unknown origin has occurred
- `ERESERVED` – the operation is refused because the FCM (or, in the case of a DCM, one of the FCMs involved in the DCM operation) is reserved by another software element and the invoking software element (possibly a secondary client) is not allowed to perform this

operation

- ENOT_IMPLEMENTED – the receiver of a HAVi message does not implement the optional API (see section 5.1.5) indicated by the Operation Code contained within the message
- EINVALID_PARAMETER – one or more parameters in a HAVi message contain invalid values
- ERESOURCE_LIMIT – the operation failed due to resource limitations at the destination device
- EPARAMETER_SIZE_LIMIT – one or more parameters in a HAVi message exceed their safe parameter size limit and the receiver is unable to handle the parameter(s)
- EINCOMPLETE_MESSAGE – the length of a HAVi message is shorter than the length required for compliant messages (using the Operation Code contained within the message)
- EINCOMPLETE_RESULT – one or more out parameters in a HAVi message are correct but incomplete. Note that this may only occur when one or more parameters are at least the safe parameter size.
- ELOCAL – the caller of a “local” API (as indicated in the “Services Provided” tables) is not on the same device as the provider of the API
- ESTANDBY – the operation is refused because the target device is in standby state

The error code appearing in the Status value returned by a HAVi API is either: one of the general codes listed above, a Messaging System error code, or an API-specific error code (one that is listed in the “Error codes” section following the description of the API).

If the Status value returned by a HAVi API contains one of the “general error codes” listed above (including SUCCESS), the API code shall be that used in invoking the API, otherwise it shall be the API code associated with the contained error (as identified in Annex 11.7). If the contained error is not listed in the "Error codes" section following the description of the API or the contained error has an invalid API code, the client of the API shall interpret the contained error as EUNIDENTIFIED_FAILURE. Therefore, if the client is a Java client, the corresponding message-sending method of the client class (see section 7.3.8.1.1), server helper class (see section 7.3.8.1.2) or the SoftwareElement class shall throw HaviUnidentifiedFailureException in these cases.

5.1.4 Parameter Size and Resource Limitations

Some of the APIs in the following sections have specifications that would allow unbounded sizes for some parameters. However, each FAV and IAV will only have a limited amount of memory. These limitations can differ from controller to controller and thus hamper interoperability between controllers.

Therefore, for variable sized (input or output) parameters in HAVi APIs a “safe parameter size limit” is specified. Such limits indicate that compliant software elements will be able to handle messages where the size of the parameter in question is less than or equal to the safe parameter size limit. However, accepting parameters of size larger than the safe parameter size limit is allowed.

The safe parameter size limit puts a requirement to support the indicated parameter size at both sending and receiving sides. At the receiving side (in parameters for servers, out parameters for clients) this means being able to receive and handle. At the sending side (out parameters for servers, in parameters for clients) this means being able to construct and send.

The server may return the EPARAMETER_SIZE_LIMIT error if it cannot handle the request due to the safe parameter size of an in parameter being exceeded.

The server shall return the EINCOMPLETE_RESULT error if the parameters it returns are valid but incomplete. Note that a server may only return this error when one or more of the parameters it returns are at least the safe parameter size.

The server shall return ERESOURCE_LIMIT if it fails to process a request due to lack of resources. If the server generates an incomplete or potentially incomplete response, i.e., one where values of the out parameters are valid but may be incomplete, this error shall not be returned.

A controller may present a notification to the user if a resource limitation problem occurs.

5.1.5 Optional APIs

Certain APIs, in particular those of specific FCMs such as the Tuner or VCR, are optional. A software element which receives a message for an unimplemented optional API shall return an error of the form ENOT_IMPLEMENTED. Only those APIs which list such an error are optional. Optional APIs are highlighted in gray in the "Services Provided" tables.

5.1.6 Vendor and Third Party Extensions

Several of the identifiers appearing in the HAVi APIs have reserved ranges for use by this and future versions of the HAVi specification. The following table summarizes these identifiers and gives rules for third party and vendor extensions.

Table 9. Extensions to HAVi Entities

Entity	IDL Type	HAVi range	Remarks
ProtocolType	uchar	0x0 – 0x7f	Annex 11.1
AttributeName	uint	0x0 – 0x7fff ffff	Annex 11.2
SoftwareElementType	uint	0x0 – 0x7fff ffff	Annex 11.3
SEID.swHandle	ushort	0x0 – 0x00ff	Annex 11.4
ApiCode	ushort	0x0 – 0x7fff	Annex 11.5
OperationId	uchar	0x0 – 0x7f	Annex 11.6
ErrorCode	ushort	0x0 – 0x7fff	Annex 11.7
AttributeIndicator	ushort	0x0 – 0x7fff	Annex 11.8
EventId	union {…}	SystemEventId	Annex 11.9
MediaFormatId	struct {…}	VendorId == 0	Annex 11.10
StreamTypeId	struct {…}	VendorId == 0	Annex 11.11
TransmissionFormat	union {…}	no extensions	Section 3.7.3
ImageTypeId	struct {…}	VendorId == 0	Annex 11.13
TransportType	ushort	no extensions	Annex 11.14
DdiElementId	ushort	0x0 – 0x7fff	Annex 11.15
OptAttrType	ushort	0x0 – 0x7fff	Annex 11.16
CompOperation	ushort	no extensions	Annex 11.17

5.1.7 Guidelines for API Updates in HAVi Versions

To avoid the introduction of new methods or system events for the smallest changes of an API, some exceptions are allowed to the general rule which states that no API may ever be updated (see section 2.8). However an extension of an API must ensure that the signature of previous versions is not broken.

To enable adding of enums and error codes to existing APIs, clients must be implemented in such a way that they are prepared to receive unknown error codes or enums and that they shall not fail as a result of a new enum or const. A client shall interpret a received unknown error code as EUNIDENTIFIED_FAILURE. Therefore, if the client is a Java client, the corresponding message-sending method of the client class (see section 7.3.8.1.1), server helper class (see section 7.3.8.1.2) or the SoftwareElement class shall throw HaviUnidentifiedFailureException in these cases. And Java applications should catch any HaviException to avoid termination on an unknown HaviException thrown by the HJA of a newer FAV.

An API may be updated as long as its signature does not change and it can be unmarshalled irrespective of the version. This may be the case when new members are added to the type definition of the enum parameters or union input parameters, or new error codes are added.

However, any change to the parameter list, or any change to the type definition of struct parameters or union output parameters will change the signature of the API. In these cases a new API shall be added with a new operation ID or a new system event shall be added with a new system event ID.

Moreover, even if the signature of the API is not changed, any change to the semantics or safe parameter size needs a new API with a new operation ID or system event ID.

5.2 Communication Media Manager

5.2.1 Services Provided

Service	Comm Type	Locality	Access
Cmm1394::GetGuidList	M	local	all
Cmm1394::Write	M	local	trusted
Cmm1394::Read	M	local	trusted
Cmm1394::Lock	M	local	trusted
Cmm1394::EnrollIndication	M	local	trusted
Cmm1394::DropIndication	M	local	trusted
<Client>::Cmm1394Indication	MB	local	CMM1394 (trusted)
NewDevices	E	local	CMM1394 (all)
GoneDevices	E	local	CMM1394 (all)
NetworkReset	E	local	CMM1394 (all)
GuidListReady	E	local	CMM1394 (all)

5.2.2 CMM1394 API

Cmm1394::GetGuidList

Prototype

```
Status Cmm1394::GetGuidList(
      out sequence<GUID> activeGuidList,
      out sequence<GUID> nonactiveGuidList)
```

Parameters

- `activeGuidList` – the GUID list of all active devices on the network. The safe parameter size limit is 63 `GUID` values.
- `nonactiveGuidList` – the GUID list of non-active devices on the network. The safe parameter size limit is 63 `GUID` values.

Description

Get GUID lists of both active and non-active devices on the network. The first item returned in `activeGuidList` shall be the GUID of the local node. A device is defined as active if it can process HAVi messages (IAV or FAV) or respond to commands from HAVi software elements (BAV or LAV). For FAV, IAV, or BAV devices, an SDD entry (**HAVi_Device_Status**) in the **HAVi_Unit_Directory** can be read to determine the status of the device (see section 9.10.4.6). A value of one indicates that the device is active. A value of zero indicates that the device is not active. An LAV device is considered active whenever its GUID is visible on the network.

Error codes

- `Cmm1394::ENOT_READY` – the GUID list is not available yet, the system may be updating it. This is a transient error, the client software element may retry.

Cmm1394::Write

Prototype

```
Status Cmm1394::Write(
      in GUID guid,
      in uint64 remoteOffset,
      in sequence<octet> data)
```

Parameters

- `guid` – target device's GUID.
- `remoteOffset` – target device's memory offset (the higher 16 bit is ignored).
- `data` – data block to be written to the remote device. The safe parameter size limit is 512 bytes (the 1394 asynchronous packet size for bus speeds of 100 Mbps).

Description

Initiate a 1394 asynchronous write transaction with the remote device, which is identified by `guid`. The specified data block will be written to the address specified by `remoteOffset` on the remote device. The execution of the function may experience some latency due to bus transitions between devices. The latency is upper bounded by the CMM's internal timeout mechanism.

It is permitted for the user of this API to submit the local GUID as an argument. No error is generated in this case and the operation will occur normally. Whether or not a 1394 transaction is performed is implementation dependent.

When performing a 1394 transaction and the size of `data` is four, a quadlet-write packet shall be used.

Error codes

- `Cmm1394::EADDRESS` – resp_address_error (see Table 6-11 of IEEE 1394-1995).
- `Cmm1394::EHARDWARE` – resp_data_error (see Table 6-11 of IEEE 1394-1995) or ack_data_error (see Table 6-13 of IEEE 1394-1995).
- `Cmm1394::ETYPE` – resp_type_error (see Table 6-11 of IEEE 1394-1995) or ack_type_error (see Table 6-13 of IEEE 1394-1995).
- `Cmm1394::ECONFLICT` – resp_conflict_error (see Table 6-11 of IEEE 1394-1995).

- Cmm1394::ERETRY – retry procedure error by ack_busy_X, ack_busy_A, ack_busy_B (see Table 6-13 of IEEE 1394-1995).

- Cmm1394::ESIZE – the packet is too large to be sent.
- Cmm1394::EUNKNOWN_GUID – the destination GUID is unknown
- Cmm1394::ETIMEOUT – no response packet has been received from the remote device within the internal timeout period or no valid ack code has been detected (see 6.2.5.2.1 of IEEE1394-1995). This timeout period is vendor dependent.
- Cmm1394::EBUSRESET – the request has not been completed due to a bus reset.

Cmm1394::Read

Prototype

```
Status Cmm1394::Read(
      in GUID guid,
      in uint64 remoteOffset,
      in short dataSize, out sequence<octet> data)
```

Parameters

- guid – target device's GUID
- remoteOffset – target device's memory offset (the higher 16 bit is ignored)
- dataSize – size of data (in byte) to be read back from the remote device
- data – data block read back from the remote device. The safe parameter size limit is 512 bytes (the 1394 asynchronous packet size for bus speeds of 100 Mbps).

Description

Initiate a 1394 asynchronous read transaction with the remote device specified by guid. dataSize bytes of data will be read from the remote device starting at address remoteOffset. The execution of the function may experience some latency due to bus transitions between devices. The latency is upper bounded by the CMM's internal timeout mechanism. This API may return fewer bytes than requested without returning an error.

It is permitted for the user of this API to submit the local GUID as an argument. No error is generated in this case and the operation will occur normally. Whether or not a 1394 transaction is performed is implementation dependent.

When performing a 1394 transaction and the size of data is four, a quadlet-read packet shall be used.

Error codes

- Cmm1394::EADDRESS – resp_address_error (see Table 6-11 of IEEE 1394-1995).
- Cmm1394::EHARDWARE – resp_data_error (see Table 6-11 of IEEE 1394-1995) or ack_data_error (see Table 6-13 of IEEE 1394-1995).
- Cmm1394::ETYPE – resp_type_error (see Table 6-11 of IEEE 1394-1995) or ack_type_error (see Table 6-13 of IEEE 1394-1995).
- Cmm1394::ECONFLICT – resp_conflict_error (see Table 6-11 of IEEE 1394-1995).
- Cmm1394::ERETRY – retry procedure error by ack_busy_X, ack_busy_A, ack_busy_B (see Table 6-13 of IEEE 1394-1995).

- Cmm1394::EUNKNOWN_GUID – the destination GUID is unknown
- Cmm1394::ETIMEOUT – no response packet has been received from the remote device within the internal timeout period or no valid ack code has been detected (see 6.2.5.2.1 of IEEE1394-

1995). This timeout period is vendor dependent.
- Cmm1394::EBUSRESET – the request has not been completed due to a bus reset.

Cmm1394::Lock

Prototype

```
Status Cmm1394::Lock(
      in GUID guid,
      in uint64 remoteOffset,
      in ushort extCode, in short dataSize,
      inout sequence<octet> oldData,
      in sequence<octet> newData)
```

Parameters

- guid – target device's GUID
- remoteOffset – target device's memory offset (the higher 16 bit is ignored)
- extCode – extended transaction code for the lock operation, which has been defined and described in the 1394 standard on Table 6-10 (P158) and Table 7-5 (P188) , respectively
- dataSize – total size of both oldData and newData (in bytes)
- oldData – data block whose size depends on the extCode. The safe parameter size limit is 8 bytes.
- newData – data block whose size depends on the extCode. The safe parameter size limit is 8 bytes.

Description

Initiate a 1394 lock transaction on the specified address in the remote device. The operation is specified by extCode. oldData is used as a verifying argument for a given lock operation. The execution of the function may experience some latency due to bus transitions between devices. The latency is upper bounded by the CMM's internal timeout mechanism though.

It is permitted for the user of this API to submit the local GUID as an argument. No error is generated in this case and the operation will occur normally. Whether or not a 1394 transaction is performed is implementation dependent.

Error codes

- Cmm1394::EADDRESS – resp_address_error (see Table 6-11 of IEEE 1394-1995).
- Cmm1394::EHARDWARE – resp_data_error (see Table 6-11 of IEEE 1394-1995) or ack_data_error (see Table 6-13 of IEEE 1394-1995).
- Cmm1394::ETYPE – resp_type_error (see Table 6-11 of IEEE 1394-1995) or ack_type_error (see Table 6-13 of IEEE 1394-1995).
- Cmm1394::ECONFLICT – resp_conflict_error (see Table 6-11 of IEEE 1394-1995).
- Cmm1394::ERETRY – retry procedure error by ack_busy_X, ack_busy_A, ack_busy_B (see Table 6-13 of IEEE 1394-1995).

- Cmm1394::ESIZE – the data size of either oldData or newData is out of the scope of any lock operation.
- Cmm1394::EUNKNOWN_GUID – the destination GUID is unknown
- Cmm1394::ETIMEOUT – no response packet has been received from the remote device within the internal timeout period or no valid ack code has been detected (see 6.2.5.2.1 of IEEE1394-1995). This timeout period is vendor dependent.
- Cmm1394::EBUSRESET – the request has not been completed due to a bus reset.

Cmm1394::EnrollIndication

Prototype

```
Status Cmm1394::EnrollIndication(
      in GUID guid,
      in OperationCode opCode,
      in uint64 offsetStart,
      in uint64 offsetEnd,
      out boolean conflicts)
```

Parameters

- `guid` – GUID of the remote device this handler has interest in.
- `opCode` – the `OperationCode` provided by the caller. This is the value that the CMM will place in the operation code of the notification message it sends to a client when an indication is found to be targeted to this client
- `offsetStart`, `offsetEnd` – specify the starting and ending addresses of the memory space the caller wants to receive and process an indication message (the higher 16 bit is ignored).
- `conflicts` – has the value `True` if the enrollment conflicts with an existing enrollment, `False` otherwise

Description

This function allows a CMM client to enroll an indication handler with specific offset range and interested GUID of a remote device. If the GUID parameter has the special "wildcard" value 0xffff ffff ffff ffff, i.e. all bits set to 1, then the CMM client is attempting to enroll an indication handler with specific offset range for all remote devices on the network.

The CMM saves the SEID of the caller (obtained from the Messaging System) along with the GUID and address range parameters. When it receives an indication from a remote device, the CMM dispatches the indication based on the memory offset value and length of the incoming packet and the `guid` of the remote device. If the address range specified by the memory offset value and length in the indication package is contained within the range specified by `offsetStart` and `offsetEnd`, and the remote device's GUID matches the GUID specified by the client, the CMM will dispatch the indication to this client through the Messaging System. If any part of the address range specified in the indication package is outside of the range specified by `offsetStart` and `offsetEnd`, the CMM does not send the indication to the corresponding client. If the GUID specified by the client is 0xffff ffff ffff ffff and if the memory offset value and length in the indication package is contained within the range specified by `offsetStart` and `offsetEnd`, then the CMM will dispatch the indication to this client for an indication received from any device on the network. The CMM sends the indication message to the client through the `<Client>::Cmm1394Indication` API. If the CMM cannot find a matching client for a given indication, it sends the 1394 response code `resp_address_error` to the remote device.

There can be multiple enrollments for the same packet. The CMM will sequentially dispatch the packet to all clients with matching enrollments via the `Cmm1394Indication` API. If one client returns `SUCCESS` the CMM sends a 1394 response using the associated `responseData`. If one client returns a 1394 error then the CMM sends a 1394 response using this as the 1394 response code. If all clients return `ENOT_INTERESTED` then the CMM sends a 1394 response using the 1394 `resp_address_error` response code. The CMM will stop dispatching to clients when it has made a 1394 response.

Note – this mechanism can lead to overlapping enrollments. However it is assumed that 1394 protocols are designed such that overlapping enrollments do not lead to conflicts.

Enrolled indications are persistent through bus resets, though it is possible that some read, write or lock notifications may be missed during a bus reset period. When a client is removed, the CMM is responsible for removing its indication handlers by monitoring the Messaging System's `MsgLeave` event. When a remote device is removed from the system, the CMM is responsible for removing all event handlers enrolled for the removed device.

Error codes

- `Cmm1394::EINVALID_OFFSET` – `offsetStart`, `offsetEnd` indicate an invalid address range for the FAV or IAV.
- `Cmm1394::EGUID_NOT_EXIST` – specified GUID does not exist on the network.

Cmm1394::DropIndication

Prototype

```
Status Cmm1394::DropIndication(
      in GUID guid,
      in uint64 offsetStart,
      in uint64 offsetEnd)
```

Parameters

- `guid` – GUID of the remote device this handler has interested in.
- `offsetStart`, `offsetEnd` – specify the starting and ending addresses of the memory space the caller wants to stop receiving and processing an indication message for (the higher 16 bit is ignored).

Description

Remove an indication handler from the CMM. The indication handler that was installed by this caller with the GUID and memory space matching the specified values will be identified and removed.

Error codes

- `Cmm1394::ENOT_FOUND` – the indication handler specified is not found.

<Client>::Cmm1394Indication

Prototype

```
Status <Client>::Cmm1394Indication(
      in GUID guid,
      in octet tcode,
      in ushort extCode,
      in uint64 memOffset,
      in ushort dataSize,
      in sequence<octet> indicationData,
      out sequence<octet> responseData)
```

Parameters

- `guid` – GUID of the remote device this handler has interest in.
- `tcode` – transaction code defined by 1394 standard (Table 6-9, P157). Typical transaction codes are block read/write and lock operations.
- `extCode` – extended transaction code for the lock operation, which has been defined and described in the 1394 standard on Table 6-10 (P158) and Table 7-5 (P188) , respectively. This field is used only when the `tcode` is a lock operation, and is ignored for any other operations.

- `memOffset` – memory location to be accessed.
- `dataSize` – the number of bytes to be read, written or locked. For a write or lock this equals the length of `indicationData`, for a read this equals the length of the `responseData` parameter to be returned.
- `indicationData` – the data portion of the incoming 1394 indication packet from the remote device. The safe parameter size limit is 512 bytes (the 1394 asynchronous packet size for bus speeds of 100 Mbps).
- `responseData` – the data portion of the 1394 response packet for the received indication. The safe parameter size limit is 512 bytes (the 1394 asynchronous packet size for bus speeds of 100 Mbps).

Description

This client API will be invoked by the CMM when an indication is identified for the client. The operation code used by the CMM to send this message comes from the previous invocation of `Cmm1394::EnrollIndication` by the client. The client processes the received indication, if it can handle the indication it returns the data for the 1394 response back to the CMM, otherwise it returns `Cmm1394::ENOT_INTERESTED`.

If the indication was a read or lock operation and `Status` is `SUCCESS`, the data provided in the `responseData` field shall be put in the 1394 response message by the CMM. If the indication was a write operation and `Status` is `SUCCESS`, the `responseData` field may be left empty and will not be used by the CMM.

Error codes

- `Cmm1394::EADDRESS` – resp_address_error (see Table 6-11 of IEEE 1394-1995).
- `Cmm1394::EHARDWARE` – resp_data_error (see Table 6-11 of IEEE 1394-1995) or ack_data_error (see Table 6-13 of IEEE 1394-1995).
- `Cmm1394::ETYPE` – resp_type_error (see Table 6-11 of IEEE 1394-1995) or ack_type_error (see Table 6-13 of IEEE 1394-1995).
- `Cmm1394::ECONFLICT` – resp_conflict_error (see Table 6-11 of IEEE 1394-1995).

- `Cmm1394::ENOT_INTERESTED` – the client does not handle the packet.

5.2.3 CMM1394 Private API

This section is informative and is not required for a HAVi compliant implementation.

The CMM may also need to provide topology, speed maps, and other environment descriptions to its clients. The topology map depicts the connectivity among physical devices. For example, it can be used to build a graphical interface that helps the user understand how devices are connected and how certain features may be used. The speed map provides information on the possible maximum speed that can be used for data transmission between any two devices in the network. It can be used to analyze the current network connection scheme, and give the user helpful suggestions for improving the performance of devices by rearranging connections. The topology map and speed map may change constantly because of dynamic insertion or removal of devices. To identify the right “version” for these maps, the bus generation number may be used. A new bus generation signifies the change of bus configuration.

Bus generation number, topology map, and speed map will remain unchanged between bus resets. To improve efficiency, the CMM may optionally cache these internally and update them only when bus reset occurs.

Cmm1394::GetBusGenerationNumber

Prototype

```
Status Cmm1394::GetBusGeneration(out long busGenNumber)
```

Parameters

- `busGenNumber` – the current bus generation number

Description

Get the current bus generation number from the network. The bus generation number specifies the number of times the current 1394 bus manager has generated topology or speed maps since its last power reset. Change of the bus generation number implies the change of network configuration.

Error codes

- `Cmm1394::ENOT_READY` – the bus generation number is not yet available, the system may be updating it.

Cmm1394::GetSpeedMap

Prototype

```
Status Cmm1394::GetSpeedMap(
      out long busGenNumber,
      out sequence<octet> speedMap)
```

Parameters

- `busGenNumber` – the current bus generation number.
- `speedMap` – speed map data block.

Description

Get the current speed map.

Error codes

- `Cmm1394::ENOT_READY` – the speed map is not yet available, the system may be updating it.

Cmm1394::GetTopologyMap

Prototype

```
Status Cmm1394::GetTopologyMap(
      out long busGenNumber,
      out sequence<octet> topMap)
```

Parameters

- `busGenNumber` – the current bus generation number
- `topMap` – pointer to an array of integer numbers, each representing a self_id packet from a 1394 device

Description

Get the current topology map.

Error codes

- `Cmm1394::ENOT_READY` – the topology map is not yet available, the system may be updating it.

5.2.4 CMM1394 Events

NewDevices

Prototype

```
void NewDevices(in sequence<GUID> guidList)
```

Parameters

- `guidList` – the GUID list of all newly connected devices. The safe parameter size limit is 62 `GUID` values.

Description

`NewDevices` is a local event. This event is generated when a new device is connected to the home network. When this happens, a network reset is triggered. The CMM gathers the GUID list of all the newly connected devices and then invokes the Event Manager to post the `NewDevices` event. The GUID list is passed to the Event Manager as additional information of the event. Since each FAV or IAV has its own CMM and all CMM are automatically activated whenever there is a network reset, the `NewDevices` event is only delivered locally within the device where the CMM resides.

GoneDevices

Prototype

```
void GoneDevices(in sequence<GUID> guidList)
```

Parameters

- `guidList` – the GUID list of all disconnected devices. The safe parameter size limit is 62 `GUID` values.

Description

`GoneDevices` is a local event. This event is generated when devices are disconnected from the home network. When this happens, a network reset is triggered. The CMM gathers the GUID list of all the disconnected devices and then invokes the Event Manager to post the `GoneDevices` event. The GUID list is passed to the Event Manager as additional information of the event. Since FAV or IAV has its own CMM and all CMM are activated whenever there is a network reset, the `GoneDevices` event is only delivered locally within the device where the CMM resides.

NetworkReset

Prototype

```
void NetworkReset()
```

Description

`NetworkReset` is a local event. This event is generated whenever there is a bus reset on the 1394. CMM may gather sequential bus resets in a short period into one `NetworkReset` event.

As opposed to the `NewDevices` and `GoneDevices` events, the CMM does not gather GUID list

of the changed devices. This event is intended for target software elements that are only interested in knowing when bus reset has occurred but are not interested in specifics of the change.

GuidListReady

Prototype

```
void GuidListReady(
   in sequence <GUID> activeGuidList,
   in sequence <GUID> nonactiveGuidList,
   in sequence <GUID> goneDevices,
   in sequence <GUID> newDevices)
```

Parameters

- `activeGuidList` – the GUID list of all active devices on the network. The first item shall be the GUID of the local node. The safe parameter size limit is 63 GUID values.
- `nonactiveGuidList` – the GUID list of non-active devices on the network. This is empty if there is no non-active device. The safe parameter size limit is 63 GUID values.
- `goneDevices` – the GUID list of all disconnected devices. This is empty if there is no gone device. The safe parameter size limit is 62 GUID values.
- `newDevices` – the GUID list of all newly connected devices. This is empty if there is no new device. The safe parameter size limit is 62 GUID values.

Description

`GuidListReady` is a local event. This event is generated when Cmm1394 becomes available for `Cmm1394::GetGuidList` API after bus reset. Cmm1394 may return `Cmm1394::ENOT_READY` for `Cmm1394::GetGuidList` API after `NetworkReset` until `GuidListReady`. The definition of `activeGuidList` and `nonactiveGuidList` contents are same as defined in `Cmm1394::GetGuidList` API.

5.3 Messaging System

5.3.1 Services Provided

Service	Comm Type	Locality	Access
MsgCallback	CB		
MsgOpen	PC		all
MsgClose	PC		all
MsgIsTrusted	PC		all
MsgGetSystemSeid	PC		all
MsgWatchOn	PC		all
MsgWatchOff	PC		all
Msg::Ping	M	global	all
MsgSendSimple	PC		all
MsgSendReliable	PC		all
MsgSendRequest	PC		all
MsgSendResponse	PC		all

MsgSendRequestSync	PC		all
SystemReady	E	global	Messaging System (all)
MsgLeave	E	global	Messaging System (all)
MsgTimeout	E	global	Messaging System (all)
MsgError	E	global	trusted

5.3.2 Messaging System Data Structures

TransferMode

Definition

```
enum TransferMode {SIMPLE, RELIABLE};
```

Description

Used to characterize the mode of a message transfer request.

ProtocolType

Definition

```
typedef octet ProtocolType;
const ProtocolType HAVi_RMI = 0x00;
```

Description

Indicates the format of a `MessageBody` (see section 3.2.1.2.4).

5.3.3 Messaging System API

MsgCallback

Prototype

```
Status MsgCallback(in ProtocolType protocol,
      in SEID sourceId,
      in SEID destId,
      in Status state,
      in sequence<octet> buffer)
```

Parameters

- `protocol` – indication of the format of the `MessageBody` part of the `buffer` parameter
- `sourceId` – the 80-bit software element identifier of the software element that issued the message
- `destId` – the 80-bit software element identifier of the target software element
- `state` – the status of the message. It may take following values:
  - `SUCCESS` if everything worked fine
  - `Msg::EALLOC` if the Messaging System cannot deliver the entire received message due to a lack of resources
  - `Msg::EDEST_SEID` if the supervision of the software element (described in `sourceId`) has been detected as disappeared. The software element `sourceId` is no longer

reachable (device unplugged, or software element performed a `MsgClose`).
Fields other than `sourceId` are undefined.

- `buffer` – consists of the `MessageBody` field in the "General Message Format" (see Figure 10)

Description

This function is the callback supplied by a software element. This call back is invoked by the Messaging System each time an incoming message (incoming reliable request or simple message) is received for that software element. It may also be invoked to notify the software element about the disappearance of a target software element (this service is provided only after a `MsgWatchOn` request).

After the callback returns, and depending on the return code (`SUCCESS` or `Msg::EFAIL`), the Messaging System acknowledges the message: if the callback returns with `SUCCESS`, the Messaging System generates a `msg_reliable_ack` message. If the callback returns with `Msg::EFAIL`, the Messaging System generates a `msg_reliable_noack` message containing `TARGET REJECT` as `MessageBody` (see 3.2.1.2.5, Table 3).

Warning: a callback function is not allowed to call blocking functions. A callback always executes in the context of the Messaging System, and is not allowed to block the Messaging System. Applications should treat the callback as an interrupt.

Error codes

- `Msg::EFAIL` – the callback failed in delivering the message

MsgOpen

Prototype

```
Status  MsgOpen(
      in MsgCallback callback,
      out SEID seid)
```

Parameters

- `callback` – the call back function that the Messaging System calls when it receives a message for `seid`.
- `seid` – the 80-bit software element identifier that has been assigned to the software element.

Description

This function is called by a software element that requires the services of the Messaging System. This function provides a unique software element identifier to the software element which is to be used by the software element to register and to communicate with other software elements. This function also allows the calling software element to provide a call back function that will be used by the Messaging System when an incoming message (either a reliable request, or a simple message) has to be passed to the software element.

Every time an `MsgOpen` call is performed, a fresh SEID is generated and returned to the caller. Once this SEID is returned, the caller (a software element) can use it as an input parameter to the other Messaging System calls presented below. For these calls, the Messaging System shall check whether the SEID provided by the caller has been assigned to that caller.

The software element identifier returned by the Messaging System will be a trusted SEID or an untrusted SEID (see section 3.2.1.1.3) according to the status or the requester. How the status of the

requester is determined by the Messaging System is implementation dependent.

Once called this function allows a software element to receive messages from the other software elements on the network.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::EALLOC` – if the Messaging System was unable to allocate a software element identifier.

MsgClose

Prototype

```
Status MsgClose(in SEID seid)
```

Parameters

- `seid` – the 80-bit software element identifier of the software element that is leaving the system

Description

This function is used by a software element when it no longer needs the services of the Messaging System. After calling this function, the software element will not be known by the Messaging System. The Messaging System will generate a `MsgLeave` event with the SEID attached.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::ESEID` – `seid` is not valid

MsgIsTrusted

Prototype

```
Status MsgIsTrusted(in SEID seid, out boolean isTrusted)
```

Parameters

- `seid` – the 80-bit software element identifier to be checked
- `isTrusted` – the result of the request. `True` means `seid` represents a trusted software element. `False` means `seid` represents an untrusted software element.

Description

This function checks whether `seid` is associated with a trusted or an untrusted software element.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::ESEID` – the SEID is not available.

MsgGetSystemSeid

Prototype

```
Status MsgGetSystemSeid(
      in SEID seid,
      in SoftwareElementType type,
      out SEID systemSeid)
```

Parameters

- `seid` – the SEID of a software element within the same node as the requested system software element
- `type` – a constant value identifying the system software element type (see Annex 11.3)
- `systemSeid` – the SEID of the requested system software element

Description

This function obtains the SEID of a system software element (see Annex 11.3) located on the same host as the specified software element (`seid` parameter) and of the specified software element type. This function does not guarantee the existence of the desired system software element.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::EELEMENT` – `type` is not a system software element type

MsgWatchOn

Prototype

```
Status MsgWatchOn(
      in SEID sourceId,
      in SEID destId)
```

Parameters

- `sourceId` – the 80-bit software element identifier of the element making the request
- `destId` – the 80-bit software element identifier of the element to be supervised

Description

This function is used to establish a "supervision" for monitoring whether a software element is present on the home network.

After calling `MsgWatchOn`, the Messaging System shall immediately check whether the target software element exists by sending a `Msg::Ping` to the Messaging System where `destId` is located. If the ping fails, `MsgWatchOn` returns an error.

To provide this service the Messaging System subscribes to events allowing it to detect whether `destId` is no longer present on the home network. These events are: `SystemReady`, `MsgLeave`, and `NetworkReset`. The Messaging System will inform `sourceId` of the "disappearance" of `destId` (by invoking `sourceId`'s callback with `state` set to `Msg::EDEST_SEID`) under the following conditions:

- the Messaging System receives a `MsgLeave` event as a result of `destId` performing a `MsgClose`
- the Messaging System receives a `NetworkReset` event and detects, by invoking `Cmm1394::GetGuidList`, that the device on which `destId` is located is no longer active
- the Messaging System receives a `SystemReady` event and `destId` referred to a non-system software element located on the device generating the event
- The Messaging System receives a MsgTimeout event and destId located on the device caused the timeout.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational

- Msg::EBUSY – the sending message was refused because the number of messages exceeded the outstanding message limit for the same source SEID or because of any other message traffic reason. Outstanding message limit is implementation dependent
- Msg::EALLOC – the supervision establishment is not possible because of local resource limitations
- Msg::ESOURCE_SEID – the source SEID is not available for this caller
- Msg::EDEST_SEID – any of the following occurred:
    - The messaging system of the target can not find any destination software element according to the destination SEID.
    - the Messaging System detected that the target software element disappeared since the Messaging System received a MsgLeave event while the Messaging System was waiting for an ACK or NOACK message, related to the Msg::Ping request
- Msg::ESUPER_EXISTS – a supervision has already been established for the destination SEID by the source SEID
- Msg::EACK – any of the following occurred:
    - ack not received to the Msg::Ping
    - the Messaging System received a NetworkReset and detected, by invoking Cmm1394::GetGuidList, that the device on which the destination software element is located is no longer on the network
    - the Messaging System received a GoneDevice or GuidListReady event and detected that the device on which the destination software element is located is no longer on the network
    - while waiting for an ACK or NOACK related to the Msg::Ping request, received a SystemReady event and detected that the device on which the destination non-system software element is located was initialized or reset
- Msg::ETIMEOUT – the Msg::Ping timed out
- Msg::EDEST_UNREACHABLE – any of the following occurred:
    - the Msg::Ping cannot be delivered because a prior message to the destination Messaging System has timed out
    - while waiting for an ACK or NOACK related to the Msg::Ping request, the Messaging System received a MsgTimeout event and detected that the device on which the destination software element is located entered an anomalous state

MsgWatchOff

Prototype

```
Status  MsgWatchOff(
      in SEID sourceId,
      in SEID destId)
```

Parameters

- sourceId – the 80-bit software element identifier of the element making the request
- destId – the 80-bit software element identifier of the element supervised

Description

This function is used to stop a supervision of destId. If the Messaging System has no supervision established for a software element, it will no longer monitor the availability of that software element.

After MsgWatchOff, if the previously supervised object disappears, the calling object will not be

informed.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::EUNKNOWN` – if the supervision is unknown or no longer established
- `Msg::ESOURCE_SEID` – the source SEID is not available for this caller

Msg::Ping

Prototype

```
Status  Msg::Ping(in SEID target)
```

Parameters

- `target` – the 80-bit software element identifier of a software element being supervised

Description

`Msg::Ping` is sent by a software element to a Messaging System to query whether `target` is present. If target is present on the remote device then `Status` indicates `SUCCESS`.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::EFAIL` – if the target is not present

MsgSendSimple

Prototype

```
Status MsgSendSimple(in ProtocolType protocol,
      in SEID sourceSeid,
      in sequence<SEID> destSeidList,
      in sequence<octet> buffer)
```

Parameters

- `protocol` – indication of the format of the `buffer` parameter
- `sourceSeid` – The software element identifier of the source software element
- `destSeidList` – the list of software element identifiers of destination elements
- `buffer` – the data to send, consists of the `MessageBody` field in the "General Message Format" (see Figure 10)

Description

This function is used to send a message to one or several destination software elements. The function returns immediately. There is no guaranty that the message will be received by all destination components. This function supports private protocols only.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::EALLOC` – the Messaging System is unable to allocate resources
- `Msg::ESEND` – if the Messaging System could not send the message to one or more of the destinations
- `Msg::EBUSY` – the sending message was refused because the number of messages exceeded the outstanding message limit for the same source SEID or because of any other message

traffic reason. Outstanding message limit is implementation dependent.
- Msg::ESOURCE_SEID – the source SEID is not available for this caller
- Msg::EPROTOCOL – attempt to send a message using a HAVi reserved protocol
- Msg::ESIZE – the message has been refused or aborted by the Messaging System to avoid a timeout.
- Msg::EDEST_UNREACHABLE – the message was not delivered to all destinations because a prior message to one or more destination Messaging Systems has timed out.

MsgSendReliable

Prototype

```
Status MsgSendReliable(in ProtocolType protocol,
      in SEID sourceSeid,
      in SEID destSeid,
      in sequence<octet> buffer)
```

Parameters

- protocol – indication of the format of the buffer parameter
- sourceSeid – the software element identifier of the source software element
- destSeid – the software element identifiers of the destination element
- buffer – the data to send, consists of the MessageBody field in the "General Message Format" (see Figure 10)

Description

This function is used to send a message to one destination software element. The function returns once the sending is completed (message acknowledge received). This service guaranties the requester that the destination software element has received the message. This function supports private protocols only.

Error codes

- Msg::ENOT_READY – the Messaging System is not operational
- Msg::EALLOC – the Messaging System is unable to allocate resources
- Msg::ESEND – if the Messaging System failed in sending the message
- Msg::EBUSY – the sending message was refused because the number of messages exceeded the outstanding message limit for the same source SEID or because of any other message traffic reason. Outstanding message limit is implementation dependent.
- Msg::ESOURCE_SEID – the source SEID is not available for this caller
- Msg::EACK – any of the following occurred:
    - ack not received
    - the Messaging System received a NetworkReset and detected, by invoking Cmm1394::GetGuidList, that the device on which the destination software element is located is no longer on the network
    - the Messaging System received a GoneDevice or GuidListReady event and detected that the device on which the destination software element is located is no longer on the network
    - while waiting for an ACK or NOACK related to the current reliable message, received a SystemReady event and detected that the device on which the destination software element is located was initialized or reset
- Msg::EOVERFLOW – the message has been sent to the target system, but required a memory allocation that was too large for the target system
- Msg::EDEST_SEID – any of the following occurred:
    - The messaging system of the target can not find any destination software

element according to the destination SEID.
        - while waiting for an ACK or NOACK related to the current reliable message, the Messaging System received a MsgLeave event, indicating that the target software element has disappeared.
- Msg::ETARGET_REJECT – the destination SEID reject the message
- Msg::EPROTOCOL – attempt to send a message using a HAVi reserved protocol
- Msg::ESIZE – the message has been refused or aborted by the Messaging System to avoid a timeout.
- Msg::EDEST_UNREACHABLE – any of the following occurred:
        - the message cannot be delivered because a prior message to the destination Messaging System has timed out
        - while waiting for an ACK or NOACK related to the current reliable message, the Messaging System received a MsgTimeout event and detected that the device on which the destination software element is located entered an anomalous state

MsgSendRequest

Prototype

```
Status MsgSendRequest(in SEID sourceSeid,
      in SEID destSeid,
      in OperationCode opCode,
      in sequence<octet> buffer,
      out uint transactionId)
```

Parameters

- sourceSeid – the software element identifier of the source software element
- destSeid – the software element identifier of the destination element
- opCode – the operation code invoked by the requester
- buffer – the request data to send (according to the prototype of the invoked method), consists of the data after the TransactionId field in the "Function Call Mapping" (see Figure 14)
- transactionId – this value allows the requester to match the future response (if any) with this request

Description

This function is used to send a "function call" message (see 3.2.3.2) to one destination software element. The Messaging System will always use the reliable mode to send the message. Therefore, the function returns once the sending is completed (message acknowledge received). This service guaranties the requester that the destination software element has received the message.

Error codes

- Msg::ENOT_READY – the Messaging System is not operational
- Msg::EALLOC – the Messaging System is unable to allocate resources
- Msg::ESEND – if the Messaging System failed in sending the message
- Msg::EBUSY – the sending message was refused because the number of messages exceeded the outstanding message limit for the same source SEID or because of any other message traffic reason. Outstanding message limit is implementation dependent.
- Msg::ESOURCE_SEID – the source SEID is not available for this caller
- Msg::EACK – any of the following occurred:
        - ack not received
        - the Messaging System received a NetworkReset and detected, by invoking Cmm1394::GetGuidList, that the device on which the destination software

element is located is no longer on the network
    - the Messaging System received a `GoneDevice` or `GuidListReady` event and detected that the device on which the destination software element is located is no longer on the network
    - while waiting for an ACK or NOACK related to the current reliable message, received a `SystemReady` event and detected that the device on which the destination software element is located was initialized or reset
- `Msg::EOVERFLOW` – the message has been sent to the target system, but required a memory allocation that was too large for the target system
- `Msg::EDEST_SEID` – any of the following occurred:
    - The messaging system of the target can not find any destination software element according to the destination `SEID.`
    - while waiting for an ACK or NOACK related to the current reliable message, the Messaging System received a `MsgLeave` event, indicating that the target software element has disappeared.
- `Msg::ETARGET_REJECT` – the destination SEID rejects the message
- `Msg::ESIZE` – the message has been refused or aborted by the Messaging System to avoid a timeout.
- `Msg`::`EDEST_UNREACHABLE` – any of the following occurred:
    - the message cannot be delivered because a prior message to the destination Messaging System has timed out
    - while waiting for an ACK or NOACK related to the current reliable message, the Messaging System received a `MsgTimeout` event and detected that the device on which the destination software element is located entered an anomalous state

MsgSendResponse

Prototype

```
Status MsgSendResponse(in SEID sourceSeid,
      in SEID destSeid,
      in OperationCode opCode,
      in TransferMode transferMode,
      in Status returnCode,
      in sequence<octet> buffer,
      in uint transactionId)
```

Parameters

- `sourceSeid` – the software element identifier of the source software element
- `destSeid` – the software element identifier of the destination element
- `opCode` – the operation code invoked by the requester
- `transferMode` – the mode (reliable or not) used to send the response
- `returnCode` – the status code returned according to the result of the operation
- `buffer` – the response data to send (according to the prototype of the invoked method), consists of the data after the 32-bit `reserved` field in the “Function Return Mapping” (see Figure 15)
- `transactionId` – this value allows the requester to match the response to be sent with the request previously received

Description

This function is used to send a “function response” message (see 3.2.3.3) to one software element which has previously performed a `MsgSendRequest` or `MsgSendRequestSync` call. According to the chosen transfer mode, the function returns immediately or once the sending is completed

(message acknowledge received).

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::EALLOC` – the Messaging System is unable to allocate resources
- `Msg::ESEND` – if the Messaging System failed in sending the message
- `Msg::EBUSY` – the sending message was refused because the number of messages exceeded the outstanding message limit for the same source SEID or because of any other message traffic reason. Outstanding message limit is implementation dependent.
- `Msg::ESOURCE_SEID` – the source SEID is not available for this caller
- `Msg::EACK` – any of the following occurred while in the reliable mode:
    - ack not received
    - the Messaging System received a `NetworkReset` and detected, by invoking `Cmm1394::GetGuidList`, that the device on which the destination software element is located is no longer on the network
    - the Messaging System received a `GoneDevice` or `GuidListReady` event and detected that the device on which the destination software element is located is no longer on the network
    - while waiting for an ACK or NOACK related to the current reliable message, receives a `SystemReady` event and detected that the device on which the destination software element is located was initialized or reset
- `Msg::EOVERFLOW` – the message has been sent to the target system, but required a memory allocation that was too large for the target system (only in reliable mode)
- `Msg::EDEST_SEID` – any of the following occurred while in the reliable mode:
    - The messaging system of the target can not find any destination software element according to the destination `SEID.`
    - while waiting for an ACK or NOACK related to the current reliable message, the Messaging System receives a `MsgLeave` event, indicating that the target software element has disappeared.
- `Msg::ETARGET_REJECT` – the destination SEID rejected the message (only in reliable mode)
- `Msg::ESIZE` – the message has been refused or aborted by the Messaging System to avoid a timeout.
- `Msg`::`EDEST_UNREACHABLE` – any of the following occurred:
    - the message cannot be delivered because a prior message to the destination Messaging System has timed out
    - while waiting for an ACK or NOACK related to the current reliable message, the Messaging System receives a `MsgTimeout` event indicating that the device on which the destination software element is located entered an anomalous state

MsgSendRequestSync

Prototype

```
Status MsgSendRequestSync(in SEID sourceSeid,
      in SEID destSeid,
      in OperationCode opCode,
      in long timeout
      in sequence<octet> bufferIn,
      out sequence<octet> bufferOut,
      out Status returnCode)
```

Parameters

- `sourceSeid` – the software element identifier of the source software element

- destSeid – the software element identifier of the destination element
- opCode – the operation code invoked by the requester
- timeout – the maximum period (in milliseconds) to wait for a response after sending the request. A value of zero defaults to the system timeout value (see section 3.2.1.2.2)
- bufferIn – the request data to send (according to the prototype of the invoked method), consists of the data after the TransactionId field in the "Function Call Mapping" (see Figure 14)
- bufferOut – the received response data (according to the prototype of the invoked method), consists of the data after the 32-bit reserved field in the "Function Return Mapping" (see Figure 15)
- returnCode – the Status code that is the result of the requested operation

Description

This function is used by a software element when it wants to send a "function call" message (see 3.2.3.2) to one destination software element and block until the response is received (synchronous mode, see section 3.2.3.5). All requests are sent in "reliable mode". A timeout value of zero defaults to the system timeout value.

The specified timeout holds for the complete duration of the API call, that is, the time between the start of the request and the reception of the response. This implies that the standard 30 second timeout for ack/noack messages is overruled when the timeout parameter is less than 30000. If the Messaging System times out while waiting for an ack/noack message and the timeout period is less than 30 seconds, then the MsgTimeout event will not be generated. Also, when the ack/noack message is received in time, but the response message is not received in time, then the MsgTimeout event shall not be generated.

Error codes

- Msg::ENOT_READY – the Messaging System is not operational
- Msg::EALLOC – the Messaging System is unable to allocate resources
- Msg::ESEND – if the Messaging System failed in sending the message
- Msg::EBUSY – the sending message was refused because the number of messages exceeded the outstanding message limit for the same source SEID or because of any other message traffic reason. Outstanding message limit is implementation dependent.
- Msg::ESOURCE_SEID – the source SEID is not available for this caller
- Msg::EACK – any of the following occurred:
    - ack not received
    - the Messaging System received a NetworkReset and detected, by invoking Cmm1394::GetGuidList, that the device on which the destination software element is located is no longer on the network
    - the Messaging System received a GoneDevice or GuidListReady event and detected that the device on which the destination software element is located is no longer on the network
    - while waiting for an ACK or NOACK related to the current reliable message, received a SystemReady event and detected the device on which the destination software element is located was initialized or reset
- Msg::EOVERFLOW – the message has been sent to the target system, but required a memory allocation that was too large for the target system
- Msg::EDEST_SEID – any of the following occurred:
    - The messaging system of the target can not find any destination software element according to the destination SEID.
    - while waiting for an ACK or NOACK related to the current reliable message, the Messaging System received a MsgLeave event, indicating that the target software element has disappeared.

- `Msg::ETARGET_REJECT` – the destination SEID rejected the message
- `Msg::ETIMEOUT` – timeout has expired
- `Msg::ESIZE` – the message has been refused or aborted by the Messaging System to avoid a timeout.
- `Msg`::`EDEST_UNREACHABLE` – any of the following occurred:
    - the message cannot be delivered because a prior message to the destination Messaging System has timed out
    - while waiting for an ACK or NOACK related to the current reliable message, the Messaging System received a `MsgTimeout` event and detected that the device on which the destination software element is located entered an anomalous state

5.3.4 Messaging System Private API

This section is informative and is not required for a HAVi compliant implementation.

MsgSysOpen

Prototype

```
Status  MsgSysOpen(
      in MsgCallback callback,
      inout SEID seid)
```

Parameters

- `callback` – the call back function that the Messaging System calls when it receives a message for `seid`.
- `seid` – as input the field contains the well known identifier of the system component. As output the Messaging System will deliver the network SEID (the GUID part of `seid` is updated).

Description

This function is called by a *system element* that requires the services of the Messaging System. This function provides a unique software element identifier to the system element which is to be used by the system element to register and to communicate with other software elements. This function allows the calling system element to provide a call back function that will be used by the Messaging System when an incoming message (either a reliable request, or a simple message) has to be passed to the system element.

Once called this function allows a software element to receive messages from the other software elements on the network.

If more than one call back is installed for a system software component then the Messaging System will only use the most recently installed call back.

Error codes

- `Msg::ENOT_READY` – the Messaging System is not operational
- `Msg::EALLOC` – if the Messaging System was unable to allocate resources

5.3.5 Messaging System Events

SystemReady

```
void SystemReady()
```

This event is generated by the Messaging System when all local system components are registered and operational. It can also be used to detect that the software element identifiers of components installed in the initialized device have probably changed (see section 3.2.1.2).

All Messaging Systems that receive this event shall abort outstanding requests to components on the device where this event originates (since those components may have been reset or have disappeared and may not respond to previously sent requests).

MsgLeave

```
void MsgLeave(in SEID seid)
```

This event is generated by the Messaging System when an element leaves the Messaging System (by calling the `MsgClose` function). It can be used to detect that the component identified by `seid` has left the system.

MsgTimeout

void MsgTimeout(in SEID seid)

This event is generated by the Messaging System if sequential 2 to 3 timeouts occur during a reliable send actions. In general, this situation is to be considered anomalous, since no acknowledgement was received from a remote Messaging System within the 30 second timeout period. seid denotes the `SEID` of the destination software element.

The Messaging System that detects 2 to 3 times successive timeout for messages to same destination device shall refuse to send messages to any software element on that device (In particular, the `MsgTimeout` event shall not be sent to the device responsible for the problem). In that case, the destination device is regarded as faulty.

To realize this, the messaging system shall maintain the successive error counter for each remote device. (If the messaging system detects `NetworkReset` or `SystemReady` event, it should clear the error counter.)

The Messaging System shall return the error code `EDEST_UNREACHABLE` for all send attempts to those software elements until the next `NetworkReset` or `SystemReady` event is received. Note successive timeout means that no ack nor noack is received in the period of that successive timeout, and no `NetworkReset` or `SystemReady` event is detected in that period.

The successive error counter will not be incremented by the Messaging System if the ack/noack timeout period was overridden by a value smaller than 30 seconds in a `MsgSendRequestSync` call (see also section 3.2.1.2.3).

MsgError

```
void MsgError(in SEID seid,
   in ushort attempts, in Status error)
```

This event enables a software element to indicate to other software elements that it has detected a persistent failure of a Messaging System. It should be posted by a software element that has made several attempts to send a message to software element `seid`, but during the period in which message sending is attempted the Messaging System persistently returns an error code other than `EACK` or `ETIMEOUT`. It is recommended that, before posting this event, at least 10 sends are attempted with at least a one second interval between sends. The number of failed attempts is recorded in `attempts` and `error` denotes the last error received. Posting this event should be interpreted as a serious system failure and indicates that the system may no longer operate as specified.

As with `MsgTimeout`, this event is intended to notify the user of a problem. There will be no corrective action by the system itself. The user may need to reset or power off the device where the software element `seid` is located. The software element posting the event is free to choose which action to take after it has posted the event. It may choose to abort its activity, to retry sending, or to take alternative actions. If the local Messaging System is not operational then the event may not be broadcast. It is recommended that a notification or warning be presented to the user.

It is recommended that a software element that handles this event checks whether the event is posted by a trusted software element. It may choose to only handle the event if this is the case.

5.4 Event Manager

5.4.1 Services Provided

Service	Comm Type	Locality	Access
EventManager::Subscribe	M	local	all
EventManager::Unsubscribe	M	local	all
EventManager::Replace	M	local	all
EventManager::AddEvent	M	local	all
EventManager::RemoveEvent	M	local	all
EventManager::PostEvent	M	local	all
EventManager::ForwardEvent	M	global	Event Manager
<Client>::EventManagerNotification	MB	local	Event Manager (all)

A software element registers with the Event Manager using its SEID and the list of events it is interested in. The Event Manager will then use the SEID to send event notification messages to the software element via the Messaging System when the events of interest occur.

5.4.2 Event Manager Data Structures

EventId

HAVi has a predefined set of system event types that are generated by trusted components, see Annex 11.9. Each event has an `EventId` – a quantity that identifies the event type to which it belongs. An `EventId` has the following structure.

```
enum EventIdSchema {SYSTEM, VENDOR, APPLICATION};

union EventId switch (EventIdSchema) {
```

```
   case SYSTEM:       SystemEventId   systemEid;
   case VENDOR:       VendorEventId   vendorEid;
   case APPLICATION:  AppEventId      appEid;
};
```

- System specified event IDs contain only the 16-bit base event number:

```
struct SystemEventId {
   ushort     base;
};
```

- Vendor specified event IDs contain a base event number plus a Vendor ID:

```
struct VendorEventId {
   ushort     base;
   VendorId   vendorId;
};
```

- Application specified event IDs contain a base event number plus a software element ID:

```
struct AppEventId {
   ushort     base;
   SEID       seid;
};
```

The three forms of `EventId` have different meanings. For `SystemEventId`, only the base event number is known. For `VendorEventId`, the base event number and a `VendorId` (e.g. `VendorId` of the event poster) are known. For `AppEventId`, the base event number and a SEID (e.g. SEID of the event poster) are known.

The HAVi system events, those listed in Annex 11.9, are always posted using the `SystemEventId` schema. Implementers may use any value for the base event number appearing in `VendorEventId` and `AppEventId`. The vendor or application developer has the responsibility of managing these base event numbers.

The Event Manager manipulates events represented by their `EventId`. For example, when a software element registers the list of events it is interested to "listen to", the software element must prepare the `EventId` for each event in the list and pass the list to Event Manager. Suppose the software element is interested in the events {*E1*, *E2-from-Vendor-XYZ*, *E3-from-Object-ABC*}, it must prepare a `SystemEventId` for *E1*, a `VendorEventId` for *E2*, and a `AppEventId` for *E3*. This list of three event identifiers is then passed to the Event Manager for registration.

When an Event Manager receives an event it delivers the event to all local software elements that have registered an event ID matching that of the received event. For two event IDs to match they must have the same `EventIdSchema` and `base`. If the schema is `VendorEventId`, then the `vendorId` fields must also be equal for a match to occur. If the schema is `AppEventId` then the `seid` fields must also be equal for a match to occur.

To post an event, the event poster needs to:

- construct the proper type of `EventId` (e.g., if the poster wants to disclose its vendor, it needs to construct a `VendorEventId` and put the right value in the `VendorId` field)

- optionally construct additional information regarding the event (e.g. device GUID list when posting a `NewDevices` event)

An `EventManager::PostEvent` message is then sent from the event poster to the Event Manager and results in the actual delivery of the event.

5.4.3 Event Manager API

EventManager::Subscribe

Prototype

```
Status EventManager::Subscribe(
      in sequence<EventId> eidList,
      in OperationCode opCode)
```

Parameters

- `eidList` – array of events, represented by their `EventId`, for which the software element wants to receive notification. The safe parameter size limit is 64 `EventId` values.
- `opCode` – the `OperationCode` provided by the caller. This is the value that the Event Manager will place in the operation code of the notification message it sends to a client when an event is to be delivered to this client.

Description

`EventManager::Subscribe` adds the software element (that has sent the message) to the Event Manager's internal table. A new entry is created to register the software element and the list of events it wishes to "listen to". There is no limit on the internal table size so long as the Event Manager can find enough system resources to maintain the table. When any of the events in the software element's "interested" event list occurs, the Event Manager sends a notification message to the software element. The message contains `opCode`, the `EventId` representing the event, and possibly additional information about the event. When the software element receives the event notification message, it uses the `opCode` to determine how to process the message. It is therefore the responsibility of the software element to define the operation code for its event notification message processing procedure call, and to pass it to the Event Manager at event registration time

If the list contains duplicate event identifiers there will be only one notification generated for such events.

If the software element has already registered or resources cannot be allocated to honor the registration request, an error is returned to the software element.

Error codes

- `ERESOURCE_LIMIT` – if resources cannot be allocated to register the software element
- `EventManager::EEXIST` – if the software element has already registered itself

EventManager::Unsubscribe

Prototype

```
Status EventManager::Unsubscribe()
```

Description

`EventManager::Unsubscribe` removes the software element from the Event Manager's internal table. The resources allocated for storing the software element are freed. This function is normally called when a software element is about to leave the HAVi environment or when it no longer wants to be notified of events. After this function is called, the software element will no longer receive event notification messages from the Event Manager.

If the software element has not registered with the Event Manager, an error is returned.

Error codes

- `EventManager::ENOT_FOUND` – if the software element has not registered with the Event Manager

EventManager::Replace

Prototype

```
Status EventManager::Replace(
      in sequence<EventId> eidList,
      in OperationCode opCode)
```

Parameters

- `eidList` – new array of events represented by their `EventId`, for which the software element wants to be notified when any of them is posted. This new event array replaces the old one in the Event Manager's internal table. If the new event array is not specified (i.e. `eidList` is a list of length zero), the old array in the Event Manager's internal table is not changed. The safe parameter size limit is 64 `EventId` values.
- `opCode` – the new value that the Event Manager will place in the operation code of the notification message it sends to a client when an event is to be delivered to this client. This new operation code replaces the old one in the Event Manager's internal table. If the new operation code is not specified (i.e. `opCode.operationId` is 0xff), the operation code in the Event Manager's internal table is not changed.

Description

Replace the software element's list of "interested" events by the new list (if it is specified) and the new operation code. Resources allocated for the software element's old event list and operation code are freed. The Event Manager's internal table is updated and resources allocated to use the new event list and operation code. After the new event list (if specified) and the new operation code (if specified) are installed, if any event in the new event list occurs, the Event Manager sends a `<Client>::EventManagerNotification` message to the software element. The message uses the specified operation code and contains the `EventId` representing the event, and possibly additional information about the event. When the software element receives the event notification message, it uses the operation code to determine how to process the message.

If the list contains duplicate event identifiers there will be only one notification generated for such events.

If resources cannot be allocated to honor the replacement request or if the software element has not registered with the Event Manager, an error is returned.

Error codes

- `ERESOURCE_LIMIT` – if resources cannot be allocated to install the software element
- `EventManager::ENOT_FOUND` – if the software element has not registered with the Event Manager

EventManager::AddEvent

Prototype

```
Status EventManager::AddEvent(in EventId eventId)
```

Parameters

- `eventId` – the new event, represented by its `EventId`, to be added to the software element's

"listen-to" event list.

Description

Add the new event `eventId` to the software element's list of "interested" events. The Event Manager's internal table is updated and resources allocated to accommodate the new event. If this new event occurs, the Event Manager sends a notification message to the software element so that it can then respond.

If resources cannot be allocated to add the new event to the software element's event list, an error is returned. Also if the software element has not registered with the Event Manager or if the software element has already registered for the event, an error is returned.

Error codes

- `ERESOURCE_LIMIT` – if resources cannot be allocated to add the new event
- `EventManager::ENOT_FOUND` – if the software element has not registered with the Event Manager
- `EventManager::EEXIST` – if the software element is already registered for this event

EventManager::RemoveEvent

Prototype

```
Status EventManager::RemoveEvent(in EventId eventId)
```

Parameters

- `eventId` – an event, represented by its `EventId`, to be removed from the software element's "listen-to" event list.

Description

Remove the event `eventId` from the software element's list of "interested" events. The Event Manager's internal table is updated and resources allocated for the removed event are freed. After invoking `EventManager::RemoveEvent`, the software element will no longer receive a notification message from the Event Manager when `eventId` occurs.

If the specified event is not on the software element's "interest" event list, of if the software element has not registered with the Event Manager, an error is returned.

Error codes

- `EventManager::ENOT_FOUND` – if the specified event is not found in the software element's event list or the software element has not registered with the Event Manager

EventManager::PostEvent

Prototype

```
Status EventManager::PostEvent(
      in EventId eventId,
      in boolean global,
      in sequence<octet> eventInfo)
```

Parameters

- `eventId` – the event, represented by its `EventId`, that the software element wants to post on the home network.

- `global` – a flag indicating whether the event is to be posted only locally within the device where the software element (the event poster) resides or to be posted globally to all FAV and IAV devices on the home network. When this flag is set to `False`, the event is posted locally. When this flag is set to `True`, the event is posted network-wide.
- `eventInfo` – a buffer containing additional information associated with the posted event. The type of information stored in `eventInfo` depends on the posted event. For example, if the posted event is `NewDevices` (when new devices are connected), then `eventInfo` contains the GUID list of the newly connected devices. If the posted event does not have additional information, the length of `eventInfo` is set to 0. The safe parameter size limit is 512 bytes.

Description

Post the specified event to the home network. Posting an event means notifying all "target" software elements (i.e., those including the event in their "listen-to" event list) regardless of their location on the network. Event notification messages are sent by the Event Managers to all target software elements. The event poster simply sends a message to an Event Manager indicating its intention to post the specified event. It is the responsibility of the Event Managers to ensure that all target software elements receive notification. An Event Manager sends notification messages to all local target software elements. If the event is to be delivered network-wide, the initial Event Manager (that receiving the `EventManager::PostEvent` message) also "broadcasts" the event (together with additional information if present) to all remote Event Managers on the network. Each remote Event Manager, when it receives the broadcast event, checks if any software element in its local device is interested in the event and sends those target software elements an event notification message. A remote Event Manager will not broadcast the event again to the network, it merely delivers the event locally. Only the initial Event Manager broadcasts the event to the network.

If an Event Manager fails in sending a notification message to any target software element or in broadcasting the posted event to remote Event Managers, an error is returned. However errors which are caused by devices that disappear during the `ForwardEvent` call will not result in an `EventManager::EDELIVERY` error.

This API should only allow trusted software elements to post system events.

Error codes

- `EventManager::EDELIVERY` – event delivery to one or more destinations has failed

EventManager::ForwardEvent

Prototype

```
Status EventManager::ForwardEvent(
      in SEID posterSeid,
      in EventId eventId,
      in sequence<octet> eventInfo)
```

Parameters

- `posterSeid` – SEID of the software element that initially posted the event
- `eventId` – the event, represented by its `EventId`, that the software element wants to post on the home network.
- `eventInfo` – a buffer containing additional information associated with the posted event. If the posted event does not have additional information, the length of `eventInfo` is set to 0. The safe parameter size limit is 512 bytes.

Description

`EventManager::ForwardEvent` is used by a local Event Manager to forward a posted event to a remote Event Manager. The remote Event Manager then performs local event delivery on behalf of the initiating Event Manager. Note that this function is invoked by a local Event Manager when a software element posts a global event (using the `EventManager::PostEvent` with `global` set to `True`). The local Event Manager performs the usual local event delivery, and also invokes `ForwardEvent` for each remote Event Manager on the network.

Error codes

- `EventManager::EFORWARDING` – event forwarding has failed to a software element which has registered for the event. If multiple forwardings fail, one error code shall be returned.

<Client>::EventManagerNotification

Prototype

```
Status <Client>::EventManagerNotification(
   in SEID    posterSeid,
   in EventId eventId,
   in sequence<octet> eventInfo)
```

Parameters

- `posterSeid` – SEID of event poster
- `eventId` – ID of posted event
- `eventInfo` – additional information provided with the event. The safe parameter size limit is 512 bytes.

Description

Used when the Event Manager delivers an event to a client by sending a “message back” to the client (with the operation code specified when the client registered interest in the event, e.g., using `EventManager::Subscribe`).

5.4.4 Event Manager Events

Trusted software elements, including HAVi system components, generate system events that reflect changes in the home network affecting other components. To ensure a basic level of interoperability, the set of HAVi system events is standardized. Annex 11.9 lists the system events used in the current version of the HAVi specification. This list is likely to grow with future versions. All base numbers for the system events are reserved for future specifications. Untrusted software elements are not allowed to post system events.

5.4.5 Event Manager Protocol

When an event is posted globally to all Event Managers in the home network, the following mechanism is used:

The event poster sends a `EventManager::PostEvent` message to its local Event Manager requesting the event to be posted on its behalf. The message contains the SEID of the event poster, the `EventId` of the event to be posted, whether the event is to be delivered locally or globally, and possibly additional information about the event.

The local Event Manager checks if any software element residing locally have the posted event in its “listen-to” event list. All target software elements that meet this condition will get an event

notification message from this Event Manager. The message contains the operation code selected by the target software, the SEID of event poster, the `EventId` of the posted event and possibly additional information about the event. The target software element, upon receiving the notification message, will presumably respond to the event. The Event Manager should not wait for this response (it should not use `MsgSendRequestSync` to send the notification message) but continue to send any remaining notification messages.

Local delivery of a posted event works in the following way. For every entry in its internal table, the Event Manager checks if the entry's target software element has requested notification of the posted event. If it has, the entry's SEID is used as the target software element to receive Event Manager notification messages. The exact message content depends on the posted event (see Annex 11.9 for examples). The general form of an event notification message is given by the `<Client>::EventManagerNotification` API.

For example, if the posted event is `NewDevices`, the additional `eventInfo` buffer contains the GUID list of the newly connected devices. The Event Manager's internal table entry also contains the `OperationCode` of the target software element. The SEID of the event poster is known to the Event Manager from the Messaging System. Thus an Event Manager always has enough information to construct the notification messages to be sent to target software elements. For each target software element, the Event Manager constructs one notification message. The message is then sent to the target software element that will presumably respond to the posted event. The Event Manager should not wait for this response (it should not use `MsgSendRequestSync` to send the notification message) but continue to send any remaining notification messages.

If the event is to be delivered network-wide, the local Event Manager (where the event poster resides) "broadcasts" the event to all Event Managers residing on the network. When a remote Event Manager receives the event, it performs local event delivery as described in the paragraph above.

A global event will be broadcast by an Event Manager using `EventManager::ForwardEvent`. The Event Manager may send `ForwardEvent` to other Event Managers in any order. If the Messaging System reports an unrecoverable error condition, or a timeout condition, the forwarding failed. However, an Event Manager shall attempt to forward a posted event to each Event Manager before returning from an `EventManager::PostEvent` method. If at least one message forwarding failed, this shall be reflected in the return code of `PostEvent`. However errors which are caused by devices having disappeared during the `ForwardEvent` call will not result in an `EventManager::EDELIVERY` error.

At initialization time, and after each network reset, the Event Manager asks the local CMM to get the list of active GUIDs for all the devices in the network. For each device, the value stored in the **HAVi_Device_Class** SDD field is retrieved (possibly using services provided by CMM, but the exact mechanism is implementation dependent). An active FAV or IAV device will have a local Event Manager, furthermore its SEID is known (since Event Managers have a well-known software element handle). Each Event Manager can thus construct the list of other Event Managers in the network. Devices that do not have the **HAVi_Device_Class** SDD field are considered to be LAV or BAV devices and will not have a local Event Manager. Nonactive FAV and IAV devices are considered to not have an active Event Manager.

5.5 Registry

5.5.1 Services Provided

Service	Comm Type	Locality	Access
Registry::RegisterElement	M	global	all
Registry::UnregisterElement	M	global	all
Registry::RetrieveAttributes	M	global	all
Registry::GetElement	M	global	all
Registry::MultipleGetElement	M	global	all
NewSoftwareElement	E	global	Registry (all)
GoneSoftwareElement	E	global	Registry (all)

5.5.2 Registry Data Structures

Attribute

Definition

```
struct Attribute {
   AttributeName   name;
   sequence<octet> value;
};
```

Description

The `Attribute` structure is used to characterize a software element through the `Registry::RegisterElement` method. Attribute values are formatted as described in Table 6 in section 3.4.1. The safe parameter size limit for `Attribute.value` is 106 bytes.

AttributeName

Definition

```
typedef uint AttributeName;
```

Description

HAVi Registry attributes are listed in Annex 11.2 and described in Table 7 found in section 3.4.1. The most significant bit of an attribute name is dedicated to the class of attribute as explained in section 3.4.1.

SoftwareElementType

Definition

```
const AttributeName   ATT_SE_TYPE = 0x00;
typedef uint SoftwareElementType;
```

Description

The software element type attribute represents the primary function of a software element. Each type is associated with different APIs. The software element types specified by HAVi are listed in Annex 11.3.

The software element types from 0x0 to 0x007f ffff are for use by trusted software elements. In particular, only trusted software elements can register themselves with these software element types.

If the software element is a system service, the type attribute designates the service itself: `COMMUNICATION_MEDIA_MANAGER`, `EVENT_MANAGER`, `REGISTRY`, `DCM_MANAGER`, `STREAM_MANAGER`, `RESOURCE_MANAGER`

If the software element is an FCM, the type attribute designates the type of functional component that the FCM controls: `GENERIC_FCM`, `VCR_FCM`, `TUNER_FCM`, etc. (The `GENERIC_FCM` type attribute is used by a software element that only exposes the basic FCM APIs and not any additional functionality.) The list will be extended as FCMs are introduced by new HAVi versions.

If the software element is a DCM, then the type attribute will be: `DCM`

If the software element is an Application Module, then the type attribute will be: `APPLICATION_MODULE`

If the software element provides a non-HAVi compliant API then the type attribute will be in the range from 0x8000 0000 to 0xffff ffff. Such kind of software elements are havlet, DDI controller and any other HAVi applications except Application Modules.

This is a mandatory attribute.

VendorId

```
const AttributeName ATT_VENDOR_ID = 0x01;
```

The `VendorId` data type is defined in section 5.1.2.

This is a mandatory attribute for DCM, FCM and Application Module software elements.

HUID

```
const AttributeName ATT_HUID = 0x02;
```

The `HUID` data type is defined in section 5.6.2.

This is a mandatory attribute for DCM, FCM and Application Module software elements.

TargetId

```
const AttributeName ATT_TARGET_ID = 0x03;
```

The `TargetId` data type is defined in section 5.6.2.

This is a mandatory attribute for DCM, FCM and Application Module software elements.

InterfaceId

```
const AttributeName ATT_INTERFACE_ID = 0x04;
```

The `InterfaceId` data type is defined in section 5.6.2.

This is a mandatory attribute for DCM, FCM and Application Module software elements.

DeviceClass

Definition

```
const AttributeName ATT_DEVICE_CLASS = 0x05;
enum DeviceClass{ LAV, BAV, IAV, FAV};
```

Description

This is the set of device categories (see section 2.3.3) standardized by the HAVi Architecture, values are: `LAV`, `BAV`, `IAV`, `FAV`

This is a mandatory attribute for DCM and FCM software elements.

GuiReq

```
const AttributeName ATT_GUI_REQ = 0x06;
typedef  uint  GuiReq;

const GuiReq NO_GUI      = 0x00;
const GuiReq DDI_GUI     = 0x01;
const GuiReq HAVLET_GUI  = 0x02;
```

Description

Designates the Graphical User Interface level(s) the software element supports. The attribute is a 32-bit field where each bit shows the availability of a corresponding level. The bit set to 1 means available. The following values are reserved:

- `NO_GUI`: the software element does not offer a user interface (default value).
- `DDI_GUI`: the software element is compatible with DDI protocol acting as a DDI Target.
- `HAVLET_GUI`: the software element provides an uploadable application.

This is an optional attribute. If this attribute is not registered, the value will be treated as `NO_GUI`.

MediaFormatId

Definition

```
const AttributeName ATT_MEDIA_FORMAT_ID = 0x07;
```

The `MediaFormatId` data type is defined in section 5.1.2.

Description

This attribute indicates the format of the media handled by the device or functional component. Possible values are listed in Annex 11.10.

This is an optional attribute

DeviceManufacturer

Definition

```
const AttributeName ATT_DEVICE_MANUF = 0x08;
typedef  wstring<50> DeviceManufacturer;
```

Description

Name of the manufacturer of the device or functional component associated with the software element. This is a UNICODE string representing the name of the manufacturer. Each character will be coded with two bytes. For example, “A” is coded as 0x0041. The string can be empty (of zero length) since it may be difficult to determine the name of an LAV manufacturer.

This is a mandatory attribute for DCMs and FCMs.

DeviceModel

Definition

```
const AttributeName ATT_DEVICE_MODEL = 0x09;
typedef  wstring<50> DeviceModel;
```

Description

This attribute is the model name, as provided by the manufacturer, of the device associated with a DCM or FCM.

This is an optional attribute.

SoftwareElementManufacturer

Definition

```
const AttributeName ATT_SE_MANUF = 0x0a;
typedef  wstring<50> SoftwareElementManufacturer;
```

Description

Name of the manufacturer of the software element. This is an optional attribute.

SoftwareElementVersion

Definition

```
const AttributeName ATT_SE_VERS = 0x0b;
typedef Version SoftwareElementVersion;
```

Description

Contains the version number of the software element according to the method described in section 2.8. This attribute is mandatory for DCMs, FCMs, Application Modules and system components.

AvLanguage

Definition

```
const AttributeName ATT_AV_LANG = 0x0c;
typedef  uint  AvLanguage;

const AvLanguage NO_AV_LANGUAGE = 0x00;
const AvLanguage AVC_LANGUAGE   = 0x01;
const AvLanguage CAL_LANGUAGE   = 0x02;
```

Description

Designates the proprietary AV command language the software element supports in addition to HAVi messaging. The attribute is a 32 bit field where each bit shows the availability of a corresponding AV language. The bit set to 1 means available. This is an optional attribute. If the attribute is not registered, its value will be treated as `NO_AV_LANGUAGE`.

UserPreferredName

Definition

```
const AttributeName ATT_USER_PREF_NAME = 0x0d;
typedef  wstring<16> UserPreferredName;
```

Description

A user selected name used to identify the device associated with a DCM or FCM. The string can be empty (of zero length). This attribute is mandatory for DCMs and FCMs.

SimpleQuery

Definition

```
struct SimpleQuery {
    AttributeName     attributeName;
    sequence<octet>   compareValue;
    CompOperation     compareOperation;
};
```

Description

`SimpleQuery` is used by the `Registry::GetElement` operation.

- `attributeName` – the name of the attribute whose value is to be compared with `compareValue`.
- `compareValue` – the value to be compared against the attribute. The format of `compareValue` follows that of attributes as described in Table 6 in section 3.4.2, its safe parameter size limit is 106 bytes.
- `compareOperation` – operation performed during comparison.

A simple query is used to find a Registry entry with a specific attribute and attribute value. A Registry entry satisfies a simple query if it has an attribute that satisfies the simple query, i.e., if the value of one of the attributes with name equal to `attributeName` satisfies: `value compareOperation compareValue`.

The above interpretation is designed for queries on attributes that may only occur once in a Registry entry. Moreover, it is suitable for specifying most queries on attributes that may occur multiple times in a Registry entry. E.g., a software element X may be registered with the attribute `MediaFormatId` occurring twice, once with the value `DISC__DVD` and once with the value `DISC__DVD__R`. Now, the query "`ATT_MEDIA_FORMAT_ID EQ DISC__DVD`" for finding FCMs

that can handle DVD discs will return software element X since it has one attribute satisfying the condition. However, note that this interpretation may sometimes seem "counter intuitive" for attributes that may occur multiple times. E.g., the query "ATT_MEDIA_FORMAT_ID NEQ DISC__DVD" will also return software element X, since it has one attribute satisfying the condition (although it also has one that does not satisfy the condition).

If compareOperation is ANY then an entry satisfies the query provided it has an attribute named attributeName (and compareValue is ignored). For EQU, NEQU, GT, GE, LT and LE, the query can be processed as follows: The query is evaluated byte by byte, starting with the most significant byte.

- Bytes are Different
  - If a pair of bytes differ then the result of the query is determined by comparing the two bytes using compareOperation.
- Bytes are Identical, Same Length
  - If compareValue and the attribute value are of the same length and are identical then EQU, GE, and LE succeed while NEQU, LT, and GT fail.
- Bytes are Identical, Different length
  - If compareValue and the attribute value are not of the same length, but are identical up to the length of the shorter sequence, then:

1. the attribute value has greater length: NEQU, GT, GE succeed while EQU, LT, and LE fail;

2. compareValue has greater length: NEQU, LT, LE succeed while EQU, GT, and GE fail.

For BWA, the query can be processed as follows: If the lengths (or types) differ, the query fails, otherwise perform the following: For each byte *i*, *Ai*, of the attribute value, and each byte *i*, *Bi*, of compareValue, if the *bitwise logical and* of *Ai* and *Bi* results in a non-zero value, then *Ai* BWA *Bi* succeeds, otherwise it fails. The bytes of the attribute value and compareValue are processed from the most significant byte (in big endian order, with the leftmost byte as the most significant byte) to the least significant byte. The query succeeds if for any byte *j*, *Aj* BWA *Bj* succeeds. The query fails if for all bytes *i* in attribute value and compareValue, *Ai* BWA *Bi* fails.

For BWO, the query can be processed as follows: If the lengths (or types) differ, the query fails, otherwise perform the following: For each byte *i*, *Ai*, of the attribute value, and each byte *i*, *Bi*, of the compareValue, if the *bitwise logical or* of *Ai* and *Bi* results in the value 0xff, then *Ai* BWO *Bi* succeeds, otherwise it fails. The bytes of the attribute value and compareValue are processed from the most significant byte (in big endian order, with the leftmost byte as the most significant byte) to the least significant byte. The query succeeds if for all bytes *j* in attribute value and compareValue, *Aj* BWO *Bj* succeeds. The query fails if for any byte *j*, *Aj* BWO *Bj* fails.

BoolOperation

Definition

```
enum BoolOperation {AND, OR};
```

Description

This type is used within the definition of the ComplexQuery structure.

ComplexQuery

Definition

```
struct ComplexQuery {
    union query1 switch(short) {
        case 0: SimpleQuery                 smplQuery1;
        case 1: sequence<ComplexQuery, 1>   cmplxQuery1;
    };

    union query2 switch(short) {
        case 0: SimpleQuery                 smplQuery2;
        case 1: sequence<ComplexQuery, 1>   cmplxQuery2;
    };

    BoolOperation boolOperation;
};
```

Description

`ComplexQuery` is used by the `Registry::MultipleGetElement` operation. A complex query is a boolean operation between two SEID lists obtained from either a simple query or another complex query.

The IDL sequence appearing in `query1` and `query2` should contain exactly one element. (The sequence construct is used simply because this is the only way to define a recursive data structure in IDL.) However, in order to comply with the HAVi CDR rules (which define the mapping from IDL into a bitflow), the length of this sequence (always 1) has to be marshalled (as an IDL long) before the single element of the sequence is marshalled. Safe parameter size limits of ComplexQuery are as follows: max number of SimpleQuerys in a whole `ComplexQuery` is 32, max depth of recursion in a whole `ComplexQuery` is 5.

5.5.3 Registry API

Registry::RegisterElement

Prototype

```
Status Registry::RegisterElement(
      in SEID seid,
      in sequence<Attribute> table)
```

Parameters

- `seid` – the unique identifier of the software element to be registered.
- `table` – the attribute table. The table is checked to assure attributes are valid. The safe parameter size limit is 100 `Attribute` values.

Description

This service primitive is used to add or modify a software element in the Registry database. If `seid` is already within the database then all of the old attributes associated to specified `seid` will be removed and the new attributes will be added. If `seid` is not present within the database then an entry is created for `seid` using the attributes in `table,` and a `NewSoftwareElement` event for the newly added `seid` is posted globally. Only local software elements (i.e., elements that reside on the same node as the Registry) may be registered.

For an untrusted caller the Registry checks whether or not the `seid` parameter is the SEID of the caller. It is forbidden for an untrusted caller to register an element with a SEID other than its own. For a trusted caller the `seid` parameter can be associated with any software element (however if `seid` is not on the same device as the Registry then an error is returned).

Furthermore, the Registry checks that untrusted software elements only register a `ATT_SE_TYPE` attribute that is among those possible for untrusted software elements (see Annex 11.3).

Error codes

- `ERESOURCE_LIMIT` – resource allocation error.
- `Registry::ELOCATION` – `seid` is a remote software element.
- `Registry::EATTRIBUTE_NAME` – the registration is not possible since a single valued system attribute name occurs multiple times.

Registry::UnregisterElement

Prototype

```
Status Registry::UnregisterElement(in SEID seid)
```

Parameters

- `seid` – the unique identifier of the software element to be unregistered

Description

`Registry::UnregisterElement` is used to remove a software element from the Registry database and a `GoneSoftwareElement` event for the removed `seid` is posted globally.

For an untrusted caller the Registry checks whether or not the `seid` parameter is the SEID of the caller. It is forbidden for an untrusted caller to unregister an element with a SEID other than its own. For a trusted caller the `seid` parameter can be associated with any software element.

Error codes

- `Registry::EIDENTIFIER` – the software element cannot be found in the local Registry

Registry::RetrieveAttributes

Prototype

```
Status Registry::RetrieveAttributes(
      in SEID seid,
      out sequence<Attribute> table)
```

Parameters

- `seid` – the unique identifier of a software element.
- `table` – indicates where the attribute list corresponding to the software element is to be copied. The safe parameter size limit is 100 `Attribute` values.

Description

This service primitive is used to read the attributes of the software element whose identifier is `seid`. The software element with this SEID may be located anywhere in the network (i.e., it need not reside on the same node as the Registry that initially receives the `RetrieveAttributes` request).

Error codes

- `Registry::EIDENTIFIER` – the software element cannot be found in any Registry
- `Registry::ENETWORK` – no response due to a network problem

Registry::GetElement

Prototype

```
Status Registry::GetElement(
      in  SimpleQuery query,
      out sequence<SEID> seidList)
```

Parameters

- `query` – specifies a simple boolean operation on one attribute of the Registry database. It contains the following elements as described above: `name`, `compareValue` and `operation`.
- `seidList` – the list of SEIDs of software elements which match the query.

Description

This primitive is used to get a list of software element identifiers that satisfy the query given through the `query` parameter. If the caller is remote to the Registry (i.e., resides on a different node than the Registry) then only SEIDs that are local to the Registry are returned. If the caller is local to the Registry then all SEIDs on the network (local as well as remote) that satisfy the query are returned.

Should `Registry::ENETWORK` be returned, the out parameter `seidList` contains all matching SEIDs that were found in the network.

Error codes

- `Registry::ENETWORK` – the response is not complete due to a network problem

Registry::MultipleGetElement

Prototype

```
Status Registry::MultipleGetElement(
      in  ComplexQuery query,
      out sequence<SEID> seidList)
```

Parameters

- `query` – specifies a complex query on several attributes of the Registry database.
- `seidList` – the list of SEIDs of software elements which match with criteria of the query

Description

This primitive is used to get a list of software element identifiers from Registry database satisfying a boolean operation between two queries (which in turn may involve other queries – see the definition of `ComplexQuery`).

Should `Registry::ENETWORK` be returned, the out parameter `seidList` contains all matching SEIDs that were found in the network.

Error codes

- `Registry::ENETWORK` – the response is not complete due to a network problem

5.5.4 Registry Events

NewSoftwareElement

```
void NewSoftwareElement (
      in SEID seid,
      in boolean hasHuid,
      in HUID huid)
```

This event is generated when a new entry is created (using `Registry::RegisterElement`). The SEID corresponds to that of the newly registered software element. `NewSoftwareElement` is not generated when an entry is updated. `NewSoftwareElement` is not generated for system elements (i.e., when system elements register themselves). The HUID corresponds to that of the newly registered software element (if it has one), `hasHuid` indicates if the HUID parameter is valid.

GoneSoftwareElement

```
void GoneSoftwareElement(in SEID seid)
```

This event is generated when an entry is removed from the Registry (using `Registry::UnregisterElement`). The SEID corresponds to the identifier of the unregistered software element.

5.5.5 Registry Protocol

Each Registry maintains a local database containing entries for the registered software elements located on the same device as the Registry.

When a Registry (A) receives a query from a local software element (i.e., running on the same device), it uses the Messaging System to forward the query to other Registries. The query is sent to all other Registries, and Registry A waits for the response messages from each remote Registry. (It is the responsibility of the Registry implementor to assure that the Registry will not be blocked indefinitely. For example, a timer can be armed just before sending the request to a remote Registry.) When all the replies have been received, then Registry A sends a response to the calling software element. The response contains all the SEIDs found in the different replies.

If one reply contains a `EINCOMPLETE_RESULT` error, this error is returned to the calling software element. If one of the remote Registries could not be reached, the return code is `Registry::ENETWORK` (however, if the forwarding fails due to the device containing the remote Registry being removed then this error is not returned). In the case where both occur, the return code passed to the calling software element may be either `EINCOMPLETE_RESULT` or `Registry::ENETWORK`.

When a Registry receives a query from a remote Registry, it examines its local database and generates a response.

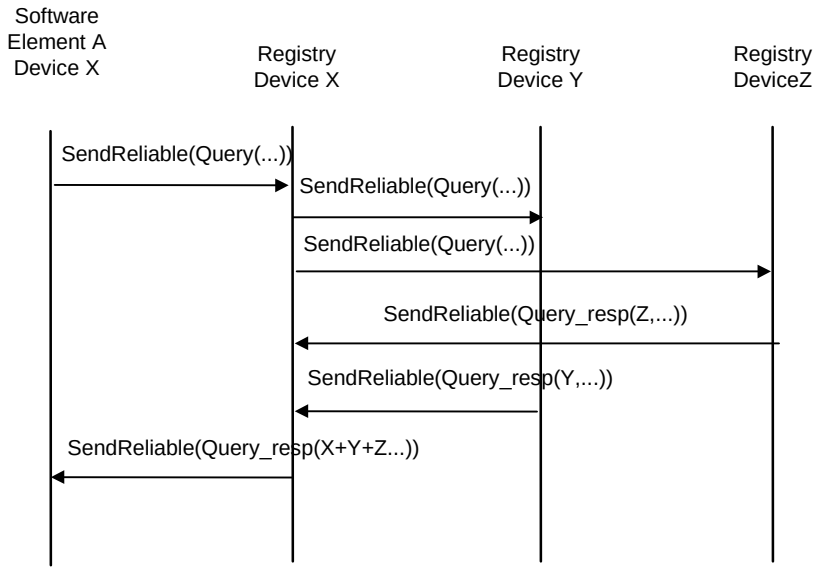

Figure 31. Registry Protocol

In a similar way, if a Registry receives a `RetrieveAttributes` request for an SEID that is not local to the Registry, the "local" Registry uses the SEID to find out on which node it resides, then forwards the retrieval request to the Registry on that remote node. The remote Registry sends the retrieval result to the "local" Registry which in turn returns it to the caller.

The `RetrieveAttributes` is only sent to the remote registry on which the requested SEID resides. If that Registry is (currently) not in the network, the `Registry::EIDENTIFIER` error is returned.

5.6 Device Control Module

This section provides a set of commands for device management within the architecture provided by a DCM. These DCM commands include areas such as connection management, informational and status queries for the device and its plugs, etc. The DCM corresponding to a device is the target of these messages. This document describes the common set of DCM commands. Regardless of the type of device that is represented by the DCM, the DCM command set must be supported. However, (proprietary) extensions of the DCM command set are allowed.

5.6.1 Services Provided

Service	Comm Type	Locality	Access	Resv Prot
Dcm::GetDeviceIcon	M	global	all	
Dcm::GetHuid	M	global	all	
Dcm::GetFcmCount	M	global	all	
Dcm::GetFcmSeidList	M	global	all	
Dcm::GetDeviceClass	M	global	all	
Dcm::GetDeviceManufacturer	M	global	all	
Dcm::GetUserPreferredName	M	global	all	

Service	Comm Type	Locality	Access	Resv Prot
Dcm::SetUserPreferredName	M	global	all	
UserPreferredNameChanged	E	global	DCM (all)	
Dcm::GetPowerState	M	global	all	
Dcm::SetPowerState	M	global	all	yes
PowerStateChanged	E	global	DCM (all)	
PowerFailureImminent	E	global	DCM (all)	
Dcm::NativeCommand	M	global	all	yes
Dcm::GetControlCapability	M	global	all	
Dcm::GetHavletCodeUnitProfile	M	global	all	
Dcm::GetHavletCodeUnit	M	global	all	
Dcm::GetPlugCount	M	global	all	
Dcm::GetPlugStatus	M	global	all	
Dcm::Connect	M	global	Stream Manager	yes
Dcm::Disconnect	M	global	Stream Manager	yes
Dcm::GetConnectionList	M	global	all	
DeviceConnectionAdded	E	global	DCM (all)	
DeviceConnectionDropped	E	global	DCM (all)	
DeviceConnectionChanged	E	global	DCM (all)	
Dcm::GetChannelUsage	M	global	all	
Dcm::GetPlugUsage	M	global	all	
Dcm::SetIecBandwidthAllocation	M	global	Stream Manager	yes
BandwidthRequirementChanged	E	global	DCM (all)	
Dcm::IecSprayOut	M	global	Stream Manager	
Dcm::IecTapIn	M	global	Stream Manager	yes
Dcm::GetSupportedTransmissionFormats	M	global	all	
Dcm::GetTransmissionFormat	M	global	all	
Dcm::SetTransmissionFormat	M	global	Stream Manager	yes
TransmissionFormatChanged	E	global	DCM (all)	
Dcm::GetContentIconList	M	global	all	
Dcm::SelectContent	M	global	all	yes
Dcm::StopContent	M	global	all	yes
ContentListChanged	E	global	DCM (all)	
Dcm::ScheduleReservation	M	global	Resource Manager	
Dcm::UnscheduleReservation	M	global	Resource Manager	
Dcm::GetScheduledActionReferences	M	global	all	
InvalidScheduledAction	E	global	DCM (all)	
Dcm::AddVirtualFcm	M	local	trusted	
Dcm::RemoveVirtualFcm	M	local	virtual FCM	

Service	Comm Type	Locality	Access	Resv Prot
Dcm::GetAvailableStreamTypes	M	global	all	
Dcm::GetStreamType	M	global	all	
Dcm::SetStreamTypeId	M	global	Stream Manager	yes
StreamTypeChanged	E	global	Dcm(all)	

5.6.2 Device Control Module Data Structures

DeviceIcon

Definition

```
struct DeviceIcon {
     Label         deviceIconName;
     OptAttrList   optionals;
};
```

Description

The device icon element is returned by the DCM and DCM Manager API calls `GetDeviceIcon`. Device icons may be used to represent a device (DCM) to a human user; for example, on a network map image. The particular form of representation is a vendor device option.

For the definition and meaning of the `Label` and `OptAttrList` types see section 5.12 APIs for Data Driven Interaction.

The `deviceIconName` label denotes the user preferred name as discussed in the DCM and Registry sections. The user preferred name can be changed via `Dcm::SetUserPreferredName`. The DCM takes care that the `deviceIconName` in the `DeviceIcon` retrieved via `Dcm::GetDeviceIcon` is updated accordingly. Note that `DeviceIcon` is also used for providing a visual representation of an Application Module. In this case, the `deviceIconName` label does not mean denoting the user preferred name of the device.

Optional Attributes

- `DEVICE_ICON_BITMAP`

Optionally, a device icon may provide a bitmap representation in one or more of the optional attributes. The size of the bitmap is recommended to be a size that would allow a controller or other user interface application to easily display on the screen a number of device representations. This would allow the user to select a given device for further device control. In any case, the bitmap should be no larger that 48 x 48 pixels. Note that the device icon's bitmap is represented by an optional attribute of type `DEVICE_ICON_BITMAP` which directly contains the pixels of the bitmap. This means that when a `DeviceIcon` has been obtained, `DdiTarget::GetDdiContent` does not need to be used to obtain the pixels.

For the definition and meaning of the `DEVICE_ICON_BITMAP` see section 5.12 APIs for Data Driven Interaction.

ContentIcon

Definition

```
struct ContentIcon {
     Label         contentIconName;
     boolean       availability;
     OptAttrList   optionals;
};
```

Description

The content icon element is returned by the API call `Dcm::GetContentIconList`. Content icons may be used to represent to a human user the content that can be provided by a device. The DCM for this device will make available, using `Dcm::GetContentIconList`, a set of content icons for user selection of the DCM's AV content. When selected, using `Dcm::SelectContent`, the target device will start playing this content out .

The `availability` attribute indicates whether or not the content shown by the content icon is (= `True`) or is not (= `False`) available now.

For the definition and meaning of the `Label` and `OptAttrList` types see section 5.12 APIs for Data Driven Interaction.

Optional Attributes

- `CONTENT_ICON_BITMAP`
- `FONTSIZE`
- `PLAYBACK_DURATION`
- `RECORDED_DATETIME`
- `BROADCAST_DATETIME`

The `PLAYBACK_DURATION` attribute shows the length of time required for playing back this content.

The `RECORDED_DATETIME` attribute shows when this content was recorded and is, for example, used for A/V storage devices.

The `BROADCAST_DATETIME` attribute shows when this content will be transmitted from a broadcasting station and, for example, is used for tuner devices.

For the definition and meaning of the optional attributes see section 5.12 APIs for Data Driven Interaction.

The size of the content icon bitmap is recommended to be a size that would allow a controller or other user interface application to easily display on the screen a number of device content representations. In any case, the bitmap should be no larger that 48 x 48 pixels.

Note that the content icon's bitmap is represented by an optional attribute of type `CONTENT_ICON_BITMAP` which directly contains the pixels of the bitmap. This means that when a `ContentIcon` has been obtained, `DdiTarget::GetDdiContent` does not need to be used to obtain the pixels.

TargetId

The Target ID is the identification of a device, functional component of a device, or Application

Module inside a device, and must have the following structure:

```
enum TargetType {DCM_61883, DCM_NON61883, FCM_61883,
                 FCM_NON61883, AM };

struct TargetId {
   TargetType type;
   GUID       guid;
   uint       n1;
   ushort     n2;
};
```

The `type` indicates whether the target is an Application Module, FCM or DCM. For the latter two it also indicates whether the associated device is IEC-61883 compliant or otherwise. FAVs, IAVs and BAVs are IEC-61883 compliant and thus their DCMs and FCMs have `type` set to `DCM_61883` or `FCM_61883` respectively. The DCM or FCM for an IEC-61883 compliant LAV shall have `type` set to `DCM_61883` or `FCM_61883` respectively. The DCM or FCM for a non IEC-61883 compliant LAV shall have `type` set to `DCM_NON61883` or `FCM_NON61883` respectively.

The `guid` indicates the device which should be used for (IEEE-1394) communication (including streaming). For non-61883 devices and application modules, this is the GUID of the host.

The interpretation of the fields `n1` and `n2` should be done according to the following table:

type	GUID of:	n1	n2
`DCM_61883`	target	0xffff ffff	0xffff
`DCM_NON61883`	host	ID of non IEC-61883 compliant LAV (assigned by host)	0xffff
`FCM_61883`	target	0xffff ffff	Index of FCM (from 0x0000 to 0xfffe)
`FCM_NON61883`	host	ID of non IEC-61883 compliant LAV (assigned by host)	Index of FCM (from 0x0000 to 0xfffe)
`AM`	host	ID of Application Module (assigned by host)	0xffff

At each moment in time, a Target ID is unique in the sense that a target is represented by only one Target ID and a Target ID refers to only one target.

Notes:

- In case of non-61883 devices, the `n1` value is assigned by the host. It is the host's responsibility to assign these values such that at each moment in time a Target ID corresponds to a single target.

- For IEC-61883 devices the Target ID is unique and persistent. However, for non IEC-61883 devices a host might not be able to assign a unique and/or persistent Target ID. So, for these types of devices it may be the case that a Target ID *X* refers to a device *x* while later on it refers to another device *y*. Moreover, it may be the case that at some moment in time a device *x* is represented by a Target ID *X* while some time later it is represented by a Target ID *Y*.

- Target IDs of Application Modules are assigned by the host on which the application is going to run; they are passed to the Application Module at installation. The GUID in this Target ID is the GUID of the host, the `n1` field is a number assigned by the host in such a way that there is no other Application Module on that node with the same Target ID (the `n2` field is 0xffff). If the host assigns a Target ID to some Application Module in a persistent way, i.e. each time the Application Module is started it gets the same Target ID and that Target ID is never assigned to another Application Module, the host can indicate this to the Application Module by the `n1Uniqueness` value provided during installation (see section 7.4.2). The Application Module shall set the `n1Uniqueness` field of its HUID according to this parameter.

InterfaceId

The Interface ID is the identification of a DCM, FCM, or Application Module and must have the following structure.

```
typedef ushort InterfaceId;
```

The Interface ID is assigned per DCM and Application Module. FCMs have the same Interface ID as their associated DCM.

The Interface ID is assigned by the vendor (author) of the DCM, FCM or Application Module.

HUID

The HUID is the unique identification of a DCM, FCM, or Application Module and must have the following structure.

```
enum FCAssigner {NONE, AVC, CAL, VENDOR, DCM};

struct HUID {
   TargetId      targetId;
   InterfaceId   interfaceId;
   VendorId      vendorId;
   boolean       n1Uniqueness;
   FCAssigner    n2Assigner;
};
```

The `targetId` identifies uniquely the target (see above).

The `vendorId` contains the ID of the vendor of the DCM, FCM or Application Module. As indicated below, HUIDs have to satisfy several properties. In general, it is the vendor indicated by this Vendor ID that guaranties that the properties are met.

The `interfaceId` is a number assigned by the vendor to identify the API supported by the software element. When a device manufacturer provides different (versions of) APIs for the same device/application, it shall differentiate their HUIDs by varying their Interface IDs.

The `n1Uniqueness` field indicates whether the host could assign the Target ID `n1` field uniquely for that device, and will never use that same value for any other device. For BAVs in BAV mode, IAVs and FAVs, `n1Uniqueness` is always `True`.

The `n2Assigner` field has the value of `NONE` for the HUIDs of DCM and AM code units. For

FCMs, n2Assigner≠NONE indicates interface uniqueness and the basis for this uniqueness:

- the AV/C address determines the FCM index. The value 'n2' is regarded as a combination of upper-byte (hereafter, n2_H) and lower byte (hereafter, n2_L), where n2_H corresponds to subunit_type or Extended_subunit_type, and n2_L corresponds to subunit_ID or Extended_subunit_ID. If Extended_subunit_type is used for AV/C address and its value is less than 224, n2_H equals to the value plus 32. Otherwise, lower 5-bits of n2_H equal to the subunit_type and upper 3-bits shall be zeros. If Extended_subunit_ID is used and its value is less than 251, n2_L equals to the value plus 4. Otherwise, lower 3-bits of n2_L equal to the subunit_ID and upper 5-bits shall be zeros. Note that n2 = 0xffff is not allowed for an FCM.
- the CAL context id determines the FCM index.
- the FCM index is assigned and guaranteed by the vendor.
- the FCM index is assigned and guaranteed by the DCM.

Note: Also for FCMs of BAVs, n2Assigner is useful to indicate (FCM) persistency.

The n1Uniqueness and n2Assigner fields together also determine whether the HUID's interface can be uniquely identified by the interfaceId and vendorId fields: if a software element with HUID *H* has some API *A* and *H*.n1Uniqueness is True and *H*.n2Assigner≠NONE, then it will *always* be the case that any software element with HUID *H* has API *A*.

The above definitions lead to the following properties for HUIDs:

- The HUID of a DCM can be inferred from the HUID of any of its FCMs by replacing the n2 sub-field of the target field (targetId) by 0xffff, the type sub-field of the target field (targetId) as appropriate (DCM_61883 in case of FCM_61883 and DCM_NON61883 in case of FCM_NON61883), and the n2Assigner field by NONE.

- The GUID of the node that is needed for (IEEE-1394) communication can be inferred from the guid sub-field of the target field (targetId).

ByteRow

```
typedef sequence<octet> ByteRow;
```

Uninterpreted row of bytes. This type is used for passing native commands to a DCM or FCM (and its associated device).

The safe parameter size limit for ByteRow values is the same as that of the data parameter for Cmm1394::Read and Cmm1394::Write.

NativeProtocol

HAVi has been designed to support the embedding of non-HAVi devices in the HAVi architecture. To allow extensive control of these devices by their native command protocols, the following type is introduced:

```
enum NativeProtocol {VENDOR_PROTOCOL};
```

NativeProtocol indicates how (via which standard) a sequence of bytes should be interpreted. VENDOR_PROTOCOL is used for vendor defined command protocols. In this case, the caller must know the vendor of the DCM target device and its native command set.

ContentType

```
enum ContentType {AUDIO, AV};
```

ContentIconRef

```
struct ContentIconRef {
   uint          handle;
   ContentIcon   icon;
};
```

NO_CHANNEL

```
typedef short ChannelNumber;
const ChannelNumber NO_CHANNEL      = -1;
```

`NO_CHANNEL` is a special value returned by `Dcm::GetChannelUsage`.

DeviceConnectionDropReason

```
enum DeviceConnectionDropReason {
   DEV_CON_DROPPED_BY_NON_HAVI_REQUEST,
   IEC_BROADCASTOUT_RESTORE_FAILURE,
   IEC_BROADCAST_BROKEN
};
```

Stream Manager Types

The DCM APIs use the following types defined in the Stream Manager section:

```
Plug, DeviceConnection, IecPlug, StreamType,
TransmissionFormat
```

Resource Manager Types

The DCM APIs use the following types defined in the Resource Manager section:

```
SAReference, Command, SAConnection, SAPeriod
```

5.6.3 Device Control Module API

Dcm::GetDeviceIcon

Prototype

```
Status Dcm::GetDeviceIcon(
      out DeviceIcon icon)
```

Parameters

- `icon` – a representation for the device the DCM represents.

Description

Provides a visual representation of the device that can be displayed to the user. It is the same representation as that acquired through `DcmManager::GetDeviceIcon` for BAV devices (and possibly LAV devices).

The device icon shall be constructed from the **HAVi_User_Preferred_Name** field in the SDD of the device and from the **HAVi_Device_Icon_Bitmap** field if it is available.

Dcm::GetHuid

Prototype
```
Status Dcm::GetHuid(out HUID dcmId)
```

Parameters
- `dcmId` – the HAVi Unique ID of the DCM

Description

Returns the HAVi Unique ID of the DCM.

Dcm::GetFcmCount

Prototype
```
Status Dcm::GetFcmCount(out ushort fcmCount)
```

Parameters
- `fcmCount` – the number of FCMs related to the DCM.

Description

Returns the number of FCMs related to the device represented by the DCM.

Dcm::GetFcmSeidList

Prototype
```
Status Dcm::GetFcmSeidList(out sequence<SEID> fcmIdList)
```

Parameters
- `fcmIdList` – a list of software element identifiers of FCMs. The safe parameter size limit is 10 `SEID` values.

Description

Returns the list of software element identifiers of the FCMs related to the device represented by the DCM.

Dcm::GetDeviceClass

Prototype
```
Status Dcm::GetDeviceClass(out DeviceClass deviceClass)
```

Parameters
- `deviceClass` – the device class, see section 5.5.2.

Description

Returns the `DeviceClass` of the device represented by this DCM.

Dcm::GetDeviceManufacturer

Prototype
```
Status Dcm::GetDeviceManufacturer(out wstring name)
```

Parameters
- `name` – the name of the manufacturer. The safe parameter size limit is the maximum length of the Registry `DeviceManufacturer` data structure.

Description

Returns the name of the manufacturer of this device.

Dcm::GetUserPreferredName

Prototype
```
Status Dcm::GetUserPreferredName(out wstring name)
```

Parameters
- `name` – a string indicating the user provided name. The safe parameter size limit is the maximum length of the Registry `UserPreferredName` data structure.

Description

Provides the name that a user has given to this specific device. If no name has been assigned the string is empty. The user preferred name is also available as an attribute in the Registry.

To share this name with other HAVi devices, a device may provide the user preferred name in the field **HAVi_User_Preferred_Name** in its SDD data.

Dcm::SetUserPreferredName

Prototype
```
Status Dcm::SetUserPreferredName(in wstring name)
```

Parameters
- `name` – a string indicating the name to be assigned to the DCM. The safe parameter size limit is the maximum length of the Registry `UserPreferredName` data structure.

Description

Sets the user preferred name of the device represented by this DCM to the specified name.

The user preferred name of a device can be retrieved from several places; the DCM and the device shall ensure that each place provides the same name. This concerns:

- the name retrieved via `Dcm::GetUserPreferredName`
- the `ATT_USER_PREF_NAME` Registry attribute for the DCM
- the `deviceIconName` in the `DeviceIcon` retrieved via `Dcm::GetDeviceIcon`

- the **HAVi_User_Preferred_Name** field in the SDD of the device. In order to be compliant with IEEE1239a-2000, the implementation shall defer the SDD updating (and thus to send back `Dcm::SetUserPreferredName` response) if the SDD updating result in a generation field value used within the last 60 seconds.

Error codes

- `ERESOURCE_LIMIT` – if the DCM was unable to allocate resources (e.g., the name is too long)

Dcm::GetPowerState

Prototype

```
Status Dcm::GetPowerState(out boolean powerState)
```

Parameters

- `powerState` – the current power state of the device

Description

A device may support two power states that are visible within the HAVi network. `True` represents that the device is powered and operating normally, `False` represents a "standby" state in which operations cannot be done directly. `Dcm::GetPowerState` provides the current power state of the device.

HAVi assumes a model in which the DCM is responsible for the power state of the device. If the power state is `False` and the DCM must provide a service for which the device is required, the DCM (not the user) must wakeup the device. Note that in standby mode, the device is not completely unpowered: it can in some way be awoken by HAVi via the network. (If this is not the case, the device is not reachable from HAVi and the DCM should not be installed.)

Dcm::SetPowerState

Prototype

```
Status Dcm::SetPowerState(inout boolean powerState)
```

Parameters

- `powerState` – the desired/obtained power state of the device represented by the DCM

Description

Set the power state of the device and possibly its functional components. When the `PowerState` is `True`, `Dcm::SetPowerState` turns on the power of the device, depending on the implementation (e.g., shared power supply) this may also turn on the power of some or all of its functional components. When `powerState` is `False`, `Dcm::SetPowerState` turns off the power of the device and the power of some or all of its functional components. If any of its functional components does not support a standby mode, the power state of the respective functional component remains `True`, but the power state of all other functional components and the device are set to `False`. The power state of the device after the call has been handled is returned.

Note: If the power state of either functional component can not be changed according to the indicated rules because its FCM is reserved by an SE other than the caller, then `ERESERVED` is returned and the power state of device and all its functional components remain unchanged.

Error codes

- `ENOT_IMPLEMENTED` – if the DCM is (always) unable to change power state
- `ERESERVED` – if the power state can not be changed since any of the functional components involved is reserved.
- `Dcm::ENOT_POSS` – if the power state can not be changed due to other reasons (user intervention).

Dcm::NativeCommand

Prototype

```
Status Dcm::NativeCommand(
      in  NativeProtocol protocol,
      in  ByteRow command,
      out ByteRow response)
```

Parameters

- `protocol` – indication of the protocol of the native command, e.g. Vendor specific, AVC, CAL, etc.
- `command` – the native command to be sent to the device. The safe parameter size limit is the same as that of the data parameter for `Cmm1394::Write`.
- `response` – the response from the device. The safe parameter size limit is the same as that of the data parameter for `Cmm1394::Read`.

Description

The DCM receives a command in one of its native command protocols. `Dcm::NativeCommand` is intended for dealing with non-HAVi standards (e.g. CAL, AV/C). A native command may have side-effects on the standard HAVi DCM interface or the interface of one of its FCMs. It is the responsibility of the DCM (and its FCMs) to assure that the HAVi standard interface is not violated and to determine whether the specific native command is accepted or not. For example, a DCM may receive an AV/C "play" request for a VCR sub-unit that has a corresponding FCM. The DCM must then assure that the FCM's state remains consistent with that of the sub-unit.

Error codes

- `Dcm::ENO_PROT` – if the specified native proprietary protocol is not supported
- `Dcm::ENO_COMMAND` – if the specified command is not supported in the specified native protocol

Dcm::GetControlCapability

Prototype

```
Status Dcm::GetControlCapability(
           out boolean extControl,
           out boolean notifies)
```

Parameters

- `extControl` – `true` if external (non-HAVi) control (e.g., manual control by the user) can occur; `false` if the DCM has exclusive control over the resources of the device.
- `notifies` – `true` if the DCM is notified of external control and so maintains a consistent view of its resources; `false` otherwise.

Description

Indicates the level of control the DCM has over its device, i.e. whether other controllers can also control the device via other means, e.g., a manual control panel. Moreover, it indicates whether the DCM receives change of state notifications from the device and so maintains a consistent view of the device state.

Dcm::GetHavletCodeUnitProfile

Prototype

```
Status Dcm::GetHavletCodeUnitProfile(
      out Version version,
      out long transferSize,
      out long codeSpace,
      out long workingSpace,
      out long chunkSize)
```

Parameters

- `version` – the lowest version of the HAVi Messaging System required by this havlet.
- `transferSize` – the number of havlet code unit bytes to be transferred (i.e., the JAR file size)
- `codeSpace` – the number of bytes required for the installed havlet code unit (read-only part)
- `workingSpace` – an estimate of the number of bytes required for the working space of the installed havlet code unit (read/write part)
- `chunkSize` – the maximum number of havlet code unit bytes the DCM can send at a time

Description

Provides the various size parameters needed for determining whether the destination of the havlet code unit (for example, an FAV UI Manager) can install the havlet. This method is only supplied if the DCM indicates that it supports a havlet via the `ATT_GUI_REQ` attribute in the Registry.

Error codes

- `ENOT_IMPLEMENTED` – the DCM has no havlet code unit

Dcm::GetHavletCodeUnit

Prototype

```
Status Dcm::GetHavletCodeUnit(
   in long firstByte,
   in long lastByte,
   out sequence<octet> byteArray)
```

Parameters

- `firstByte` – the number of the first byte of the transferred havlet code unit byte array wanted
- `lastByte` – the number of the last byte of the array wanted
- `byteArray` – the byte array requested (empty if none could be delivered or if invalid `firstByte` and/or `lastByte` values are supplied). The safe parameter size limit is 512 bytes (the 1394 asynchronous packet size for bus speeds of 100 Mbps).

Description

Provides the bytecode that can be used to install and execute an havlet, a Level 2 application stored in this DCM. A havlet code unit receiver can request all or some of the bytes of the havlet code unit from the DCM. It should first use `Dcm::GetHavletCodeUnitProfile` to determine if it is capable of retrieving and installing the code unit. `firstByte` and `lastByte` should indicate

subsequent parts of the code unit to be transferred. The first byte of the code unit is number 1; the last byte is the value of `transferSize`. The amount of bytes requested (`lastByte - firstByte + 1`) shall not exceed the value of `chunkSize`. This method is only supplied if the DCM indicates that it supports a havlet via the `ATT_GUI_REQ` attribute in the Registry.

The format of the bytecode and the way it should be handled by an FAV are described in the section 7.4.

Error codes

- `EINVALID_PARAMETER` – the values of `firstByte` and/or `lastByte` are invalid
- `ENOT_IMPLEMENTED` – the DCM has no havlet code unit

Dcm::GetPlugCount

Prototype

```
Status Dcm::GetPlugCount(
      in TransportType type,
      out ushort  inCount,
      out ushort  outCount)
```

Parameters

- `type` – indication of the location of DCM plugs; i.e. 1394 plugs on the device (`type` is `IEC61883`) or external plugs on the device (`type` is `CABLE`).
- `inCount` – the number of input plugs of the specified `type`.
- `outCount` – the number of output plugs of the specified `type`.

Description

Provides information about the number of input and output plugs on the DCM, e.g. the number of 1394 plugs on the device or the number of (non-1394) device plugs to the external world.

Error codes

- `EINVALID_PARAMETER` – if the specified `TransportType` is neither `IEC61883` nor `CABLE`

Dcm::GetPlugStatus

Prototype

```
Status Dcm::GetPlugStatus(
      in Plug plug,
      out PlugStatus status)
```

Parameters

- `plug` – identifies a plug on the DCM or an FCM associated with the DCM
- `status` – the status information for the specified plug

Description

`Dcm::GetPlugStatus` returns status information for the specified plug.

Error codes

- `Dcm::ENO_ADDR` – if the specified `Plug` does not exist for this DCM or an FCM associated with the DCM

Dcm::Connect

Prototype

```
Status Dcm::Connect(
      in SEID caller,
      in Plug src,
      in Plug dest)
```

Parameters

- `caller` – the SEID of the original software element on behalf of which the connect is to be performed.
- `src` – the location of the plug that will be the source of the connection; this can be an FCM plug or a DCM plug.
- `dest` – the location of the plug that will be the destination of the connection; this can be an FCM plug or a DCM plug.

Description

Establish a device connection on the device represented by the DCM between the specified source plug and the specified destination plug. DCM and FCM plugs are numbered starting from zero, the total number of plugs can be obtained via `Dcm::GetPlugCount` and `Fcm::GetPlugCount`. The connect operation is called by Stream Managers. The reservation protection check is not done using the actual caller of `Dcm::Connect` but using the (original) caller supplied as a parameter by the Stream Manager. If this caller is the DCM itself (as results from an invocation of `Dcm::SelectContent`), then the reservation protection check is not performed. Reservation check is not performed on src.

Per breakable connection, a DCM keeps track of the Stream Managers that established the connection (via `Dcm::Connect`). In case of a `Dcm::Disconnect` call by a Stream Manager and in case a Stream Manager is removed from the network (network partition), that Stream Manager is removed from the list. A DCM can use `MsgWatchOn` to be notified of disappearing Stream Managers.

If no Stream Managers are left, the DCM can decide to break the connection or mark it as “stale”. A stale connection will always be broken in case a newly created connection would be hampered. A DCM is allowed to treat non-HAVi connections in the same way as permanent connections: connections that cannot be broken. For these non-breakable connections, the DCM does not need to maintain a list of involved Stream Managers.

Error codes

- `Dcm::ENO_ADDR` – if one of the specified `Plug` values does not exist for this DCM or an FCM associated with the DCM
- `Dcm::ENOT_SUPPORTED` – it is (always) not possible to establish the connection
- `Dcm::ENOT_POSS` – it is currently not possible to establish the connection

Dcm::Disconnect

Prototype

```
Status Dcm::Disconnect(
      in SEID caller,
      in Plug src,
      in Plug dest)
```

Parameters

- `caller` – the SEID of the original software element on behalf of which the disconnect is to be performed.
- `src` – the location of the plug that is the source of the connection to be disconnected
- `dest` – the location of the plug that is the destination of the connection to be disconnected

Description

Removes a device connection on the device represented by the DCM. DCM and FCM plugs are numbered starting from zero, the total number of plugs can be obtained via `Dcm::GetPlugCount` and `Fcm::GetPlugCount`. The disconnect operation is called by Stream Managers. The reservation protection check is not done using the actual caller of `Dcm::Connect` but using the (original) caller supplied as a parameter by the Stream Manager. If this caller is the DCM itself (as results from an invocation of `Dcm::StopContent`), then the reservation protection check is not performed. Reservation check is not performed on src.

For breakable connections: remove the Stream Manager from the list of Stream Managers that are involved in this connection. If no Stream Managers are left, the DCM can decide to break the connection or mark it as "stale". The IEC61883 broadcast connection associated with the broken or stale connection should be broken by the DCM or device itself if the Dcm or device itself has no need to continue the broadcast connection.

For non-breakable connections (permanent or non-HAVi), no action of the DCM is needed.

Error codes

- `Dcm::ENO_ADDR` – if one of the specified `Plug` values does not exist for this DCM or an FCM associated with the DCM

Dcm::GetConnectionList

Prototype

```
Status Dcm::GetConnectionList(
      in Plug plug,
      out sequence<DeviceConnection> list,
      out sequence<boolean> preemptable)
```

Parameters

- `plug` – address of the plug for which device connections are to be reported
- `list` – a list of device connections from or to the plug indicated by `plug`. The safe parameter size limit 20 `DeviceConnection` values.
- `preemptable` – indicates which of the connections in `list` the DCM will allow a Stream Manager to preempt. The safe parameter size limit 20 `boolean` values.

Description

Provides a list of all device connections in the device represented by the DCM that are either from or to the plug indicated by `plug`.

Error codes

- `Dcm::ENO_ADDR` – if the specified `Plug` does not exist for this DCM or an FCM associated with the DCM

Dcm::GetChannelUsage

Prototype

```
Status Dcm::GetChannelUsage(
      in IecPlug    plug,
      out short  channel)
```

Parameters

- `plug` – a DCM plug for the `IEC61883` transport type (a PCR).
- `channel` – the 1394 isochronous channel number.

Description

Provides the 1394 channel number that is used by the specified plug. If no channel is used by the plug, `NO_CHANNEL` is returned.

Error codes

- `Dcm::ENO_ADDR` – if the specified `IecPlug` does not exist for this DCM

Dcm::GetPlugUsage

Prototype

```
Status Dcm::GetPlugUsage(
      in ushort  channel,
      out sequence<IecPlug>    list)
```

Parameters

- `channel` – a 1394 isochronous channel number.
- `list` – a list of `IecPlug` values (plug control register numbers). The safe parameter size limit 20 `IecPlug` values.

Description

Providing a 1394 isochronous channel number, this method obtains a (possibly empty) list giving the `IecPlug` structure of plugs that use the specified channel.

Error codes

- `Dcm::ENO_ADDR` – if `channel` does not refer to a proper 1394 channel number

Dcm::SetIecBandwidthAllocation

Prototype

```
Status Dcm::SetIecBandwidthAllocation(
      in SEID    caller,
      in IecPlug plug,
      in uint    maxBandwidth)
```

Parameters

- `caller` – the SEID of the original software element on behalf of which the bandwidth allocation strategy is to be set.
- `plug` – a source DCM plug for the `IEC61883` transport type (an oPCR)
- `maxBandwidth` – the requested static payload

Description

`Dcm::SetIecBandwidthAllocation` is used to set a bandwidth allocation strategy (see section 5.9.5.4.1) for the specified oPCR. If `maxBandwidth` is 0, dynamic bandwidth allocation is requested, if `maxBandwidth` is non-zero then static bandwidth allocation is requested. The bandwidth allocation strategy can be changed only if the plug has no IEC 61883 connections (i.e., the point-to-point counter and broadcast bit are zero) and only via calls to `SetIecBandwidthAllocation`.

When dynamic bandwidth allocation is set it is the responsibility of the DCM to assure that bandwidth allocation is attempted (whether by the device or itself) whenever there is a change in bandwidth requirements of the plug. The DCM will then post the `BandwidthRequirementChanged` event.

When static bandwidth allocation is set the device shall set the payload field in the specified plug to `maxBandwidth` / (*8000 * 32*). It is the responsibility of the DCM to assure that bandwidth allocation is not attempted (whether by the device or itself) whenever there is a change in bandwidth requirements of the plug.

When static bandwidth allocation is set, bandwidth allocation may still be required if the maximum bandwidth specified is less than the `maxBandwidth` of the stream type of the source FCM plug attached to the IEC plug. In such cases a `BandwidthRequirementChanged` event is posted and indicates that the new bandwidth requirement cannot be honored because bandwidth allocation is not attempted due to the setting of static bandwidth allocation.

This API is called only by Stream Managers. The reservation protection check is not done using the actual caller of `Dcm::SetIecBandwidthAllocation` but using the (original) caller supplied as a parameter by the Stream Manager.

The Stream Manager shall call this API when establishing an IEC connection if the plug is attached to a source FCM plug and it produces a variable rate stream type. It should be called after setting the transmission format and after making internal attachments to the plug (since these actions may also result in the payload being altered).

Error codes

- `Dcm::ENO_ADDR` – if the specified `IecPlug` does not exist for this DCM
- `Dcm::EDEV_BUSY` – the point-to-point counter or broadcast bit are non-zero for the specified `IecPlug`
- `Dcm::ENOT_POSS` – if the DCM cannot assure that the payload field of the plug contains the payload value corresponding to `maxBandwidth` (either `maxBandwidth` is too large or too small or the device does not support static bandwidth allocation)

Dcm::IecSprayOut

Prototype

```
Status Dcm::IecSprayOut (
      in SEID       caller,
      in IecPlug    plug,
      in ushort        channel,
      in ushort        payload)
```

Parameters

- `caller` – the SEID of the original software element requesting the connection.
- `plug` – identifies the DCM plug to use as the source of an IEC 61883 broadcast-out

connection.
- `channel` – the value to be set for Channel number field of the oPCR.
- `payload` – the value to be set for payload field of the oPCR.

Description

Establish an IEC61883 broadcast-out connection from the specified plug. IEC 61883 allows broadcast connections to only be established by the AV device on which the PCR is located. Thus the DCM shall establish the broadcast-out connection by proprietary communication with the device (when the DCM is remote to the device). It is the responsibility of the device to establish the broadcast-out connection according to IEC 61883, including allocation of bandwidth and channel. It is also the responsibility of the device to restore the broadcast-out connection. If the restoration fails, a `DeviceConnectionDropped` event should be posted by the DCM. Dcm's are suggested to select Data rate considering capability of sink devices and the network.

When `plug.pcrNum` is `ANY_PLUG`, the DCM should return `Dcm::ENO_ADDR` or `ENOT_IMPLEMENTED` if the API is not supported.

This API is called only by Stream Managers. The reservation protection check is not done using the actual caller of `Dcm::IecSprayOut` but using the (original) caller supplied as a parameter by the Stream Manager.

Error codes

- `ENOT_IMPLEMENTED` – it is (always) not possible to establish a broadcast-out connection
- `Dcm::ENO_ADDR` – if the specified plug value does not exist for this DCM
- `Dcm::EDEV_BUSY` – unable to use the plug
- `Dcm::EINSUFF_BANDWIDTH` – bandwidth allocation has failed
- `Dcm::EINSUFF_CHANNEL` – channel allocation has failed

Dcm::IecTapIn

Prototype

```
Status Dcm::IecTapIn (
      in SEID       caller,
      in IecPlug    plug,
      in ushort     isocChannel)
```

Parameters

- `caller` – the SEID of the original software element requesting the connection.
- `plug` – identifies the DCM plug to use as the sink of an IEC 61883 broadcast-in connection.
- `isocChannel` – the 1394 isochronous channel to use for the broadcast-in connection.

Description

Establish an IEC61883 broadcast-in connection to the specified plug. IEC 61883 allows broadcast connections to only be established by the AV device on which the PCR is located. Thus the DCM shall establish the broadcast-in connection by proprietary communication with the device (when the DCM is remote to the device).

When `plug.pcrNum` is `ANY_PLUG`, the DCM should return `Dcm::ENO_ADDR` or `ENOT_IMPLEMENTED` if the API is not supported.

This API is called only by Stream Managers. The reservation protection check is not done using the actual caller of `Dcm::IecTapIn` but using the (original) caller supplied as a parameter by the

Stream Manager.

Error codes

- `ENOT_IMPLEMENTED` – it is (always) not possible to establish a broadcast-in connection
- `Dcm::ENO_ADDR` – if the specified plug value does not exist for this DCM
- `Dcm::EDEV_BUSY` – unable to use the plug

Dcm::GetSupportedTransmissionFormats

Prototype

```
Status Dcm::GetSupportedTransmissionFormats(
      in StreamType type,
      in Plug plug,
      out sequence<TransmissionFormat> formats)
```

Parameters

- `type` – a stream type
- `plug` – a DCM plug
- `formats` – list of transmission formats. The safe parameter size limit is 10 `TransmissionFormat` values.

Description

Given a specific plug and stream type, this method provides the list of transmission formats supported by that plug for use with the stream type. The first member of `formats` is the `TransmissionFormat` value returned by `Dcm::GetTransmissionFormat`.

Error codes

- `Dcm::ENO_ADDR` – if the specified `Plug` does not exist for this DCM

Dcm::GetTransmissionFormat

Prototype

```
Status Dcm::GetTransmissionFormat(
      in Plug plug,
      out TransmissionFormat format)
```

Parameters

- `plug` – a DCM plug
- `format` – a transmission format.

Description

Given a specific plug, indicated by its plug number and its direction, this method provides the transmission format currently assigned to that plug.

Error codes

- `Dcm::ENO_ADDR` – if the specified `Plug` does not exist for this DCM

Dcm::SetTransmissionFormat

Prototype

```
Status Dcm::SetTransmissionFormat(
      in SEID caller,
      in Plug plug,
      in TransmissionFormat format)
```

Parameters

- `caller` – the SEID of the original software element on behalf of which the transmission format is to be changed.
- `plug` – a DCM plug
- `format` – a transmission format

Description

Given a specific plug, indicated by its address, this method assigns the transmission format for the specified plug. The transmission format assigned must be a format supported by the plug as provided by the `Dcm::GetSupportedTransmissionFormats` primitive. Even in this case it is possible that the transmission format cannot be set (e.g., some devices may not be able to change the transmission format when the plug is active). In this case, the method will fail as indicated in the return code. This operation is called by Stream Managers. The reservation protection check is not done using the actual caller of `Dcm::SetTransmissionFormat` but using the (original) caller supplied as a parameter by the Stream Manager. If this caller is the DCM itself (as results from an invocation of `Dcm::SelectContent`), then the reservation protection check is not performed.

Error codes

- `Dcm::ENO_ADDR` – if the specified `Plug` does not exist for this DCM
- `Dcm::ENOT_SET` – it is (currently) not possible to set the transmission format to the specified value
- `Dcm::ENOT_SUPPORTED`– if it is (always) not possible to set the transmission format to the specified value.

Dcm::GetContentIconList

Prototype

```
Status Dcm::GetContentIconList(
      in ContentType type,
      out sequence<ContentIconRef> list)
```

Parameters

- `type` – content type of the streams in the content icon list.
- `list` – list of the content icons and their handles for the device. The safe parameter size limit is 2 Kbytes.

Description

Get the list of content icons for this device for either the `AUDIO` or `AV` content types. If the device does not support content icons, the list will be empty. The handles identify the content icons and will refer to the same content until the DCM disappears. *Note* – this does not mean that the content icon also needs to be available until the DCM disappears.

Dcm::SelectContent

Prototype

```
Status Dcm::SelectContent(
```

```
      in ContentType contentType,
      in uint handle,
      in boolean dynamicBw,
      in FcmPlug sink)
```

Parameters

- `contentType` – the content type of the `ContentIconList` from which content is to be selected
- `handle` – reference number of the content to be selected. This is the value of one of the handle entries in the list returned by `Dcm::GetContentIconList` and obtained with `type` set to `contentType` (the content icon is identified by this reference number).
- `dynamicBw` – indicates desired bandwidth allocation strategy
- `sink` – the sink plug to be used for the content

Description

Selects the actual content, identified by `handle`, to flow to the indicated sink. The DCM uses `StreamManager::FlowTo` to create a connection from the FCM that provides the specified content. The `ConnectionHint` can just use "any" values. A DCM should use `MsgWatchOn` to be notified when the caller disappears: the DCM should then perform `Dcm::StopContent`.

Error codes

- `Dcm::ENO_CONT` – if no content is available of the specified type for the specified handle
- `Dcm::ESINK_FCM` – the FCM indicated by `sink` does not exist
- `Dcm::ESINK_PLUG` – the FCM indicated by `sink` does not contain the specified plug
- `Dcm::ENO_CONNECTION` – the `FlowTo` connection could not be established

Dcm::StopContent

Prototype

```
Status Dcm::StopContent(
      in ContentType contentType,
      in uint handle)
```

Parameters

- `contentType` – the content type of the `ContentIconList` from which content is to be stopped.
- `handle` – reference number of the content to be selected. This is the value of one of the handle entries in the list returned by `Dcm::GetContentIconList` and obtained with `type` set to `contentType` (the content icon is identified by this reference number).

Description

Stops the flowing of the content identified by `handle`. The DCM uses `StreamManager::Drop` to break the connection from the FCM that provides the specified content.

Error codes

- `Dcm::ENO_CONT` – if no content is available of the specified type for the specified handle
- `Dcm::ENOT_RUN` – if the content indicated by `contentType` and `handle` is not currently selected

Dcm::ScheduleReservation

Prototype

```
Status Dcm::ScheduleReservation(
      in sequence<Command> startCommandsList,
      in sequence<Command> stopCommandsList,
      in sequence<SAConnection> connectionList,
      in DateTime startTime,
      in DateTime stopTime,
      in SAPeriod periodicity,
      in sequence<HUID> involvedFcmList,
      in wstring<50> userInfo,
      in long index)
```

Parameters

- `startCommandsList, stopCommandsList` – commands to be executed during the Scheduled Action. The commands are listed by order of execution. The safe parameter size limits are 1 Kbyte.
- `connectionList` – list of connections that have to be established from and by this DCM. The safe parameter size limit is 10 `SAConnection` values.
- `startTime, stopTime` – date and time information about the Scheduled Action
- `periodicity` – indicates the periodicity of the Scheduled Action. If a periodicity is specified, the Scheduled Action starts and stops every day or every week at the time indicated in `startTime`
- `involvedFcmList` – list of the involved FCMs. The safe parameter size limit is 10 `HUID` values.
- `userInfo` – string field containing the reservation reason
- `index` – index of the Scheduled Action

Description

Reservation of the involved resources (distribution of the Scheduled Action *for one DCM*), specifying all the commands that will have to be performed at start/stop time. All commands are checked in order to ensure the FCMs will perform the required commands at start time.

Implementation guideline: in addition to the schedule reservation information, the DCMs should store the Action Scheduler SEID along with the `index` parameter in order to entirely reference Scheduled Actions in which they are involved.

Error codes

- `Dcm::ECOMMANDS` – reservation is rejected because of invalid commands
- `Dcm::ECONNECTIONS` – reservation is rejected because of invalid connections
- `Dcm::ESCHED_OVERLAP` – reservation is rejected because of scheduling overlap

Dcm::UnscheduleReservation

Prototype

```
Status Dcm::UnscheduleReservation(in long index)
```

Parameters

- `index` – index of the Scheduled Action (of the calling Action Scheduler)

Description

Cancellation of the scheduled reservation.

Dcm::GetScheduledActionReferences

Prototype

```
Status Dcm::GetScheduledActionReferences(
      out sequence<SAReference> saReferenceList)
```

Parameters

- `saReferenceList` – list of Scheduled Actions references. The safe parameter size limit is 20 `SAReference` values.

Description

Gets the list of Scheduled Action references in which this DCM is involved.

Error codes

- `Dcm::ENONE` – no Scheduled Action is associated with this DCM

Dcm::AddVirtualFcm

Prototype

```
Status Dcm::AddVirtualFcm (
      out HUID fcmHuid)
```

Parameters

- `fcmHuid` – the HUID of the FCM as assigned by the DCM

Description

Results in the addition of the calling (trusted) software element to the set of FCMs associated with the target DCM. This target DCM must represent an FAV device. The target DCM will take care that in the Registry, the `ATT_SE_TYPE` attribute of the calling software element is changed to `GENERIC_FCM`.

The availability of handling isochronous streams by software depends on the system design of FAV devices. There may be DCMs for FAVs which do not support this API.

From the perspective of an application, the virtual FCM will be treated like other FCMs. In particular `Dcm::Connect` can be used to make device connections involving the virtual FCM. As with connection requests in general, not all requests may be possible for the DCM to configure. In the case of device connections involving virtual FCMs, the DCM will only allow device connections between IEC61883 input plugs and the virtual FCM. This restriction may be relaxed in future versions of HAVi.

Error codes

- `ENOT_IMPLEMENTED` – if this DCM has no support for virtual FCMs (if the DCM is not representing an FAV)
- `ERESOURCE_LIMIT` – if this DCM has insufficient resources to support this additional FCM.

Dcm::RemoveVirtualFcm

Prototype

```
Status Dcm::RemoveVirtualFcm()
```

Description

Results in the removal of the calling (trusted) FCM from the set of FCMs associated with the target DCM. This target DCM should represent an FAV device. The target DCM will take care that in the Registry, the `ATT_SE_TYPE` attribute of the calling FCM is changed from `GENERIC_FCM` to its value prior to `Dcm::AddVirtualFcm`.

Error codes

`ENOT_IMPLEMENTED` – if this DCM has no support for virtual FCMs (if the DCM does not represent an FAV)

Dcm::GetAvailableStreamTypes

Prototype

```
Status Dcm::GetAvailableStreamTypes(
      in  plug DcmPlug ,
      in  plug FcmPlug,
      out sequence<StreamType> types)
```

Parameters

- `DcmPlug` – DCM plug, specifying one end of an attachment.
- `FcmPlug`– FCM plug, specifying the other end of an attachment.
- `types` – list of stream types

Description

Given an attachment specified by `DcmPlug` and `FcmPlug`, this method provides the list of currently available stream types for the attachment (For example, some DCM may return temporary stream type even if there is no tape. Such DCM should accept `setStreamTypeId` for the stream type and may generate `StreamTypeChanged` event after stream types come concrete). The members of `types` list have unique stream type Ids and the first member of the list has the same stream type ID as the `StreamType` value returned by `Dcm::GetStreamType`.

This API returns available stream types that are available after the plugs are attached even if the plugs are not yet attached. `FcmPlug` and `DcmPlug` parameters must be the same direction, otherwise `EINVALID_PARAMETER` is returned.

Dcm::GetStreamType

Prototype

```
Status Dcm::GetStreamType(
      in  plug DcmPlug,
      out StreamType type)
```

Parameters

- `DcmPlug` – indicating DcmPlug
- `type` – stream type

Description

Given an attachment specified by `DcmPlug`, this method provides the stream type currently assigned to that attachment. The `maxBandwidth` field of `type` shall indicate the maximum bandwidth (in bps) produced by the plug (a source plug), or capable of being consumed (a sink plug).This API returns value only when there is an attachment.

Error codes

- `Dcm::ENO_ATTACH` – there is no attachment between specified plugs.
- `Dcm::ENO_ADDR` – if the specified plug does not exist for this DCM.

Dcm::SetStreamTypeId

Prototype

```
Status Dcm::SetStreamTypeId(
      in SEID caller,
      in plug DcmPlug,
      in StreamTypeId typeId)
```

Parameters

- `caller` – the SEID of the original software element on behalf of which the stream type is to be changed.
- `DcmPlug` –Dcm Plug
- `typeId` –stream type Id to be set.

Description

Given an attachment specified by DcmPlug , this method assigns the stream type ID to the specified attachment. The stream type ID assigned must be one supported by the attachment as provided by the `Dcm::GetSupportedStreamTypes` primitive. This operation is called by Stream Managers. The reservation protection check is not done using the actual caller of `Dcm::SetStreamTypeId` but using the (original) caller supplied as a parameter by the Stream Manager. If this caller is the DCM to which this FCM belongs (as results from an invocation of `Dcm::SelectContent`), then the reservation protection check is not performed. This API returns value only when there is an attachment. If the `DcmPlug` specified is sink then the reservation check is not performed.

Error codes

- `Dcm::ENO_ATTACH` – there is no attachment between specified plugs.
- `Dcm::ENO_ADDR` – if the specified plug does not exist for this DCM.
- `Dcm::ENOT_SET` – it is currently not possible to set the stream type ID to the specified value.
- `Dcm::ENOT_SUPPORTED` – it is (always) not possible to set the stream type ID to the specified value

5.6.4 Device Control Module Events

UserPreferredNameChanged

Prototype

```
void UserPreferredNameChanged(in wstring name)
```

Parameters

- `name` – the new user preferred name.

Description

Notification of a change of the user preferred name of this DCM. If a user preferred name is changed via `Dcm::SetUserPreferredName`, the DCM shall post this event after changing all the following values:

- the name retrieved via Dcm::getUserPreferredName
- the `ATT_USER_PREF_NAME` Registry attribute for the DCM
- the `deviceIconName` in the `DeviceIcon` retrieved via Dcm::GetDeviceIcon
- the **HAVi_User_Preferred_Name** field in the SDD of the device.

PowerStateChanged

Prototype

```
void PowerStateChanged(in boolean powerState)
```

Parameters

- `powerState` – the current power state of the device represented by the DCM

Description

Notification of a change in the power state of the device represented by the DCM.

PowerFailureImminent

Prototype

```
void PowerFailureImminent()
```

Description

Indication that a power failure is imminent. This event indicates that the whole device represented by the DCM, including its FCMs is about to enter an unpowered state. The consequences of entering the unpowered state are implementation specific.

DeviceConnectionAdded

Prototype

```
void DeviceConnectionAdded(
```

```
   in Plug src, in Plug dest)
```

Parameters

- `src` – identifies a plug on the DCM or an FCM associated with the DCM
- `dest` – identifies a plug on the DCM or an FCM associated with the DCM

Description

Notifies that a device connection from `src` to `dest` has been added due to external (non-HAVi) control.

Note: Some types of devices might not support this feature (see `Dcm::GetControlCapability` above).

DeviceConnectionDropped

Prototype

```
void DeviceConnectionDropped(
      in Plug src, in Plug dest,
      in DeviceConnectionDropReason reason)
```

Parameters

- `src` – identifies a plug on the DCM or an FCM associated with the DCM
- `dest` – identifies a plug on the DCM or an FCM associated with the DCM
- `reason` – the reason why the connection has become inoperable

Description

Notifies one of the following has occurred:

- a device connection has been dropped due to external (non-HAVi) control
- break of IEC 61883 broadcast-out connection due to failure in restoring after bus reset.
- break of IEC 61883 broadcast connection.

Stream Managers drop their connections that become inoperable and post `ConnectionDropped` events with the DropReason set to `DEVICE_CONNECTION_DROPPED`, `IEC_BROADCAST_BROKEN` or `RESTORE_FAILURE`. This event should not be generated when a IEC 61883 broadcast connection which is not involved in HAVi connection has been broken.

Note: Some types of devices might not support this feature (see `Dcm::GetControlCapability` above).

Note2: break of a broadcast connection does not mean drop of the attachment.

DeviceConnectionChanged

Prototype

```
void DeviceConnectionChanged(
      in Plug plug,
      in InternalPlug oldPlug, in InternalPlug newPlug,
      in StreamType newSType,
      in TransmissionFormat newTFormat)
```

Parameters

- `plug` – identifies a plug on the DCM or an FCM associated with the DCM
- `oldPlug` – identifies a plug on an FCM associated with the DCM
- `newPlug` – identifies a plug on an FCM associated with the DCM
- `newSType` – new stream type used by the connection
- `newTFormat` – new transmission format used by the connection

Description

Notifies that a device connection has changed due to device function. For example, an external (non-HAVi) action, such as pushing a button on the device's front panel may cause the change. It is also possible that the change may occur as a side-effect of HAVi APIs (e.g., `Vcr::Stop`) – depending on the implementation of the DCM, FCM and device.

A `DeviceConnectionChanged` event indicates that the device connection between `plug` and `oldPlug` has replaced by one between `plug` and `newPlug`. A `DeviceConnectionChanged` event is not accompanied by a pair of `DeviceConnectionDropped` and `DeviceConnectionAdded` events. When a `DeviceConnectionChanged` occurs there may also be a change in stream type and/or transmission format used by the source or sink, however it does not trigger `StreamTypeChanged` or `TransmissionFormatChanged` events. The Stream Managers involved (those managing connections with the specified plugs) will check the compatibility of source and sink and attempt to set the new stream type and transmission format if needed. The compatibility check is not applied to the `CABLE` connection.

Note: Some types of devices might not support this feature (see `Dcm::GetControlCapability` above).

TransmissionFormatChanged

Prototype

```
void TransmissionFormatChanged(
      in Plug          plug,
      in TransmissionFormat    format)
```

Parameters

- `plug` – a DCM plug on the node
- `format` – the new transmission format used by `plug`

Description

Posted when the transmission format of the specified plug has changed as a result of operation of the device (rather than as a result of `Dcm::SetTransmissionFormat`). The plug is a source. The Stream Managers involved (managing connections with the specified source) will check the compatibility of sinks and set the new transmission format of sinks (via `Dcm::SetTransmissionFormat`). The compatibility check is not applied to the `CABLE` connection.

BandwidthRequirementChanged

Prototype

```
void BandwidthRequirementChanged(
      in IecPlug plug, in boolean allocated)
```

Parameters

- `plug` – a source IEC plug on the node
- `allocated` – indicates whether sufficient bandwidth is allocated for the plug

Description

Posted when the bandwidth requirement of the specified plug has changed. Stream Managers that have connections involving `plug`, will check whether the current payload field in the IEC plug is still less than the `maxBandwidth` field of the stream type of the sink FCM. If the current payload is greater than this value or `allocated` is `False` a `ConnectionChanged` event is posted with change reason `BANDWIDTH_ADAPTATION_FAILURE`. The operational status of the connection becomes `FAILURE`, the failure reason is `BANDWIDTH_FAILURE`.

ContentListChanged

Prototype

```
void ContentListChanged(in ContentType type)
```

Parameters

- `type` the content type of the content icon list for which content has changed.

Description

Notification of a change in a content list of a DCM. Note that changes in content lists should not be too frequent because users should be able to select content icons from the lists presented to them.

InvalidScheduledAction

See section 5.10.4 on Resource Manager events.

StreamTypeChanged

Prototype

```
void StreamTypeChanged(
      in plug SrcDcmPlug,
      in StreamType type)
```

Parameters

- `SrcDcmPlug` -DcmPlug
- `type` – changed stream type

Description

Posted when the stream type of the specified plug has changed as a result of operation of the functional component (rather than as a result of `Dcm::SetStreamTypeId`). The plug is a source. The Stream Managers involved (managing connections with the specified source) will check the compatibility of sinks and set the new stream type at the sink (via `Dcm::SetStreamTypeId`). The compatibility check is not applied to the `CABLE` connection.

5.7 Functional Component Module

This section specifies a set of commands for control, administration and management of FCMs. Generally, FCM commands can be subdivided into the following categories:

- Commands that deal with administration and management of FCMs, similar to the commands for DCMs. These commands are supported by all FCMs, independent of their particular type.

- Commands which apply only to a specific functional domain such as VCRs or tuners. These function-specific commands typically correspond to the native commands such as, for a VCR functional component, the PLAY, STOP and REWIND commands. Because the HAVi messages for these commands must be sent to a software element within the network, the FCM also acts as the target for these messages. Proprietary extensions of function-specific command sets are allowed.

This section only deals with the first category – administration and management commands of FCMs. Function-specific command sets are described in section 6.

5.7.1 Services Provided

Service	Comm Type	Locality	Access	Resv Prot
Fcm::GetHuid	M	global	all	
Fcm::GetDcmSeid	M	global	all	
Fcm::GetFcmType	M	global	all	
Fcm::GetPowerState	M	global	all	
Fcm::SetPowerState	M	global	all	yes
PowerStateChanged	E	global	FCM (all)	
PowerFailureImminent	E	global	FCM (all)	
Fcm::NativeCommand	M	global	all	yes
Fcm::SubscribeNotification	M	global	all	
Fcm::UnsubscribeNotification	M	global	all	
<Client>::FcmNotification	MB	global	FCM (all)	
Fcm::GetPlugCount	M	global	all	
Fcm::GetSupportedStreamTypes	M	global	all	
Fcm::Wink	M	global	all	
Fcm::Unwink	M	global	all	
Fcm::CanWink	M	global	all	
Fcm::Reserve	M	global	Resource Manager	
Fcm::Release	M	global	Resource Manager	
Fcm::GetReservationStatus	M	global	all	
Fcm::GetWorstCaseStartupTime	M	global	all	
ReserveIndication	E	global	FCM (all)	
ReleaseIndication	E	global	FCM (all)	

Service	Comm Type	Locality	Access	Resv Prot
Fcm::SetPlugSharing	M	global	all	yes
PlugSharingChanged	E	global	FCM (all)	
Fcm::IecAttach	M	local	DCM	
Fcm::IecDetach	M	local	DCM	

5.7.2 Functional Component Module Data Structures

FcmAttributeIndicator

For defining notifications, the following types are needed:

```
typedef ushort FcmAttributeIndicator;
```

Indication of a specific FCM attribute, to be specified for each specific functional component type separately (see Annex 11.8).

FcmAttributeValue

```
typedef sequence<octet> FcmAttributeValue;
```

Since the type of the possible FCM attributes cannot be specified in advance, a general data structure is used (sequence of bytes). For each attribute, the range of values and their representation as a byte string is specified by the specific FCM. The CDR standard is used to map types into byte sequences. In the case of short or long values, the value is always passed in big-endian format.

The safe parameter size limit for `FcmAttributeValue` values is 32 bytes.

NotificationId

```
typedef ushort NotificationId;
```

Unique (for a specific FCM) identification of a notification type, provided by the notification source and used by the subscriber to refer to the specific subscription.

DCM Types

The FCM APIs use the following types defined in the DCM section:

```
NativeProtocol, ByteRow
```

Stream Manager Types

The FCM APIs use the following types defined in the Stream Manager section:

```
IecPlug, StreamType, Direction, Plug
```

Resource Manager Types

The FCM APIs use the following type defined in the Resource Manager section:

```
ClientRole
```

ClientRecord

Definition

```
struct ClientRecord {
   SEID            client;
   boolean         primary;
   ClientRole      clientRole;
   wstring<50>     info;
   OperationCode   preemptionRequestCode;
};
```

Description

A specification of a client record (used for FCM reservation). `client` specifies the client. `primary` is `True` for primary access rights and `False` for secondary access rights. `clientRole` specifies its user or system role. `info` specifies the client's supplied information. `preemptionRequestCode` is the operation code of the client's preemption request message.

5.7.3 Functional Component Module API

Fcm::GetHuid

Prototype

```
Status Fcm::GetHuid(out HUID fcmId)
```

Parameters

- `fcmId` – HAVi Unique ID of the FCM

Description

Returns the HAVi Unique ID of the functional component represented by the FCM.

Fcm::GetDcmSeid

Prototype

```
Status Fcm::GetDcmSeid(out SEID dcmId)
```

Parameters

- `dcmId` –software element identifier of the DCM

Description

Returns the software element identifier of the DCM that represents the device on which the functional component represented by this FCM resides.

Fcm::GetFcmType

Prototype

```
Status Fcm::GetFcmType(out SoftwareElementType type)
```

Parameters

- `type` – the software element type of the FCM as defined in Annex 11.3

Description

Returns the HAVi standardized type (`VCR_FCM`, `TUNER_FCM`, `GENERIC_FCM` etc.) of the functional component represented by this FCM. (See the description of `SoftwareElementType` in section 5.5.2. and the list of FCM types in Annex 11.3.)

Fcm::GetPowerState

Prototype

```
Status Fcm::GetPowerState(out boolean powerState)
```

Parameters

- `powerState` – the current power state of the functional component

Description

Power management of a functional component is similar to the power management of a device. A functional component may support two power states that are visible within the HAVi network. `True` represents that the functional component is powered and operating normally, `False` represents a "standby" state in which operations cannot be done directly. `Fcm::GetPowerState` provides the current power state of the functional component.

HAVi assumes a model in which the FCM is responsible for the power state of the functional component. If the power of the functional component is off and the FCM must provide a service in which the device is really required, the FCM (not the user) must wake up the device. Note that in standby mode, the functional component is not completely unpowered: it can in some way be awoken by HAVi via the network. (If this is not the case, the device is not reachable from in HAVi and the complete FCM should not be available).

Fcm::SetPowerState

Prototype

```
Status Fcm::SetPowerState(inout boolean powerState)
```

Parameters

- `powerState` – the desired/obtained power state of the device represented by the FCM

Description

Try to set the power state of the functional component to the `powerState` provided. When the `powerState` is `True`, `Fcm::SetPowerState` turns on the power of the functional component, and possibly (depending on the device itself) also the power of the whole device. When `powerState` is `False`, `Fcm::SetPowerState` turns off the power of the functional component, if possible. If the FCM represents a functional component that does not support a standby mode (or which requires the whole device to be in standby), the power state of the device remains `True`. The power state of the functional component after the call has been handled is returned.

Error codes

- `ENOT_IMPLEMENTED` – if the FCM is (always) unable to change power state
- `Fcm::ENOT_POSS` – if the FCM is unable to change power state because of one or more FCMs, which are required to change power state simultaneously, are reserved by other SE(s).

Fcm::NativeCommand

Prototype

```
Status Fcm::NativeCommand(
      in NativeProtocol protocol,
      in  ByteRow command,
      out ByteRow response)
```

Parameters

- `protocol` – indication of the native protocol of the command
- `command` – native command to be sent to the device. The safe parameter size limit is the same as that of the data parameter for `Cmm1394::Write`.
- `response` – the response from the device. The safe parameter size limit is the same as that of the data parameter for `Cmm1394::Read`.

Description

The FCM receives a command in one of its native command protocols. Useful for dealing with other non-HAVi standards (e.g. CAL, AV/C). A native command may have side-effects on the standard HAVi DCM interface or the interface of one of its FCMs. It is the responsibility of the FCM (together with its DCM and other FCMs) to assure that the HAVi standard interface is not violated and to determine whether the native command is accepted or not.

Error codes

- `Fcm::ENO_PROT` – if the specified native proprietary protocol is not supported
- `Fcm::ENO_COMMAND` – if the specified command is not supported in the specified native protocol

Fcm::SubscribeNotification

Prototype

```
Status Fcm::SubscribeNotification(
      in  FcmAttributeIndicator attributeIndicator,
      in  FcmAttributeValue value,
      in  CompOperation comparator,
      in  OperationCode opCode,
      out FcmAttributeValue currentValue,
      out NotificationId notificationId)
```

Parameters

- `attributeIndicator` – indication of an FCM state attribute.
- `value` – value of an FCM state attribute. The safe parameter size limit is that of `FcmAttributeValue`.
- `comparator` – specification of the condition on `value`, on which the event has to be generated.
- `opCode` – the `OperationCode` provided by the caller. This is the value that the FCM will place in the operation code of the notification message it sends to a client.
- `currentValue` – the current value of the FCM attribute. The safe parameter size limit is that of `FcmAttributeValue`.
- `notificationId` – identification (within an FCM) of a notification.

Description

Registers the caller as a software element interested in a setting of the specified FCM attribute. In

general, the caller will be notified when the value of the attribute identified by `attributeIndicator` and `value` satisfy `comparator`. If the comparator is `ANY`, every setting of the attribute results in a notification, and the `value` parameter is ignored.

`SubscribeNotification` returns the `notificationId` that the FCM will use in the notification of this situation. This ID must also be used by the caller to unsubscribe. It also returns the current value of the FCM state attribute.

The subscription ends when the application unsubscribes explicitly or when the application is no longer reachable by the FCM (detected by the `MsgWatchOn` facility of the FCM's local Messaging System).

This definition only specifies the general part of the API. If a specific FCM supports notification of attribute changes, the attributes and the values (of each attribute) for the FCM are specified in Annex 11.8.

The `FcmAttributeValue` in the above specification is a sequence of bytes. In the FCM API sections, specific types (specified in IDL) are associated with `FcmAttributeValue`. The mapping of the IDL types onto the sequence of bytes is again according to CDR. The first byte of `FcmAttributeValue` is considered the "zero index" for natural boundary alignment.

Error codes

- `EINVALID_PARAMETER` – the value of `value`, or some other parameter, is invalid for the FCM
- `ERESOURCE_LIMIT` – if the FCM was unable to allocate resources to register this indication listener

Fcm::UnsubscribeNotification

Prototype

```
Status Fcm::UnsubscribeNotification(
      in NotificationId notificationId)
```

Parameters

- `notificationId` – unique (within the FCM) notification ID

Description

Unsubscribes the caller as a software element that is interested in the occurrence of the situation specified by `notificationId`. The FCM will stop sending notifications to the caller.

Error codes

- `Fcm::ENO_NOT` – if no notification with this ID has been registered for the caller

<Client>::FcmNotification

Prototype

```
Status <Client>::FcmNotification(
      in NotificationId notificationId,
      in FcmAttributeIndicator attributeIndicator,
      in FcmAttributeValue value)
```

Parameters

- `notificationId` – identification (within the FCM) of the notification.
- `attributeIndicator` – indication of an FCM state attribute.
- `value` – value of an FCM state attribute. The safe parameter size limit is that of `FcmAttributeValue`.

Description

During the subscription, the FCM will send a "message back" to the subscriber, each time the situation occurs using the operation code the subscriber has specified via `Fcm::SubscribeNotification`.

Fcm::GetPlugCount

Prototype

```
Status Fcm::GetPlugCount(
      out ushort inCount,
      out ushort outCount)
```

Parameters

- `inCount` – the number of input FCM plugs
- `outCount` – the number of output FCM plugs

Description

Provides information about the number of input and output plugs on the FCM.

Fcm::GetSupportedStreamTypes

Prototype

```
Status Fcm::GetSupportedStreamTypes(
      in  ushort plugNum,
      in  Direction  direction,
      out sequence<StreamType> types)
```

Parameters

- `plugNum` – the number of a plug on the FCM.
- `direction` – the direction of the plug `plugNum` on the FCM.
- `types` – list of stream types. The safe parameter size limit is 20 `StreamType` values.

Description

Given a specific FCM plug, indicated by its plug number and its direction, this method provides the list of stream types possibly supported by that plug. The members of this list have unique stream type IDs and the first member of the list has the same stream type ID as the `StreamType` value returned by `Dcm::GetStreamType`. This API returns all supported stream types even it can't be set now (e.g., It returns a stream type even when current media is not suitable for the stream type).

Error codes

- `Fcm::ENO_ADDR` – if the specified plug (indicated by `plugNum` and `direction`) does not exist for this FCM

Fcm::Wink

Prototype
```
Status Fcm::Wink()
```

Description

The `Fcm::Wink` command is intended to attract the attention of the user. It may result in the device blinking a LED, making a sound, displaying a logo, etc. How a device responds to `Fcm::Wink` is implementation dependent.

Error codes
- `ENOT_IMPLEMENTED` – the functional component represented by the FCM is not capable of winking
- `Fcm::EWAS_WINKING` – the functional component represented by the FCM was already winking

Fcm::Unwink

Prototype
```
Status Fcm::Unwink()
```

Description

Stop the winking behavior of a device.

Error codes
- `ENOT_IMPLEMENTED` – the functional component represented by the FCM is not capable of winking
- `Fcm::EWAS_NOT_WINKING` – the functional component represented by the FCM was not winking

Fcm::CanWink

Prototype
```
Status Fcm::CanWink(out boolean canWink)
```

Parameters
- `canWink` – if true, the FCM is capable of winking

Description

Determine if a device can wink.

Fcm::Reserve

Prototype
```
Status Fcm::Reserve(
   in SEID client, in ClientRole role,
   in wstring<50> info, in boolean primary,
   in boolean nonIntrusive,
   in OperationCode preemptionRequestCode)
```

Parameters

- `client` – a contender for the resource
- `role` – the user or system role of the contender
- `info` – the information string related to this reservation
- `primary` – `True` if reservation as a primary client; `False` if reservation as a secondary client
- `nonIntrusive` – `True` if the intention of the reservation is non-intrusive; otherwise the intention is preemptive
- `preemptionRequestCode` – the operation code of the client's preemption request message

Description

An FCM reservation can only be done by a Resource Manager (the FCM shall verify this). Before issuing this request, the Resource Manager may use `Fcm::GetReservationStatus` as appropriate to retrieve the current reservation-related data from the FCM. The client will have either primary or secondary access rights if the reservation succeeds.

The `nonIntrusive` parameter prevents that accidentally a resource preemption takes place in case a non-intrusive reservation was intended. Its value therefore needs to be `True` if the client requested a non-intrusive resource reservation.

If a client has already reserved the resource, another reservation for the same client shall be accepted (unless another client has a conflicting reservation). Its access rights shall be according to the last reservation request, and its previous reservation shall be undone by an implicit release (an event `ReleaseIndication` is posted).

It shall always be possible to accept a preemptive reservation for a primary contender, unless both contender and client are of type "system". If the resource is being preempted for a primary contender, a reservation for the current primary client is undone by an implicit release. A primary reservation can be done independently of the number of accepted secondary clients, and without having to release the resource for any secondary client. (Note that there can be secondary clients without a primary client.)

If secondary access is supported by the FCM, then preemption for secondary access rights always succeeds when the contender is of type "user". If the resource cannot accept any additional secondary clients, one of the current secondary clients shall be selected for an implicit release of the resource. When the contender is of type "system" preemption for secondary access rights fails. In case the resource is preempted, its state shall remain unchanged (no return to the neutral state). If it is not possible or not valid to perform a reservation, the error code `ENO_RESERVE` is returned.

Each FCM shall use the Messaging System's watch-on facility to detect the disappearance of any of its clients. An FCM should return to a neutral state if its last client disappeared in this way (see `Fcm::Release`).

Either all three methods `Reserve`, `Release`, and `GetReservationStatus`, and the events `ReserveIndication` and `ReleaseIndication` shall be implemented by the FCM, or none of them shall be. If these methods are not supported, the error code `ENOT_IMPLEMENTED` is returned if any of them is invoked.

The execution of this API does not depend on the power state of the FCM. This means that a FCM shall not use the error code of `ESTANDBY`.

Error codes

- `Fcm::ENO_RESERVE` – the reservation failed
- `ENOT_IMPLEMENTED` – the resource does not allow reservations

Fcm::Release

Prototype

```
Status Fcm::Release(in SEID client, in boolean neutral)
```

Parameters

- `client` – a presumed current client of the resource
- `neutral` – if `True`, the FCM will return to a neutral state before it is released

Description

An `FCM::Release` can only be done by a Resource Manager (the FCM shall verify this). The data on the client and its access rights will be removed from the FCM. If a client disappears from the network, the FCM will undo (release) the client's reservation. Through the `neutral` parameter, a releasing client can determine whether or not to return the state of the FCM to *neutral*. This shall only have an effect if this was the last client of the FCM. It is up to the FCM what the meaning of this neutral state is, but its intention is that the FCM becomes "inactive" after it was released in the neutral state. A non-neutral release of FCMs allows handing over resources to other applications in a state-preserving manner.

For example, releasing a VCR FCM in a neutral state by the single client of the FCM might prevent any further recording activity on the inserted tape, despite the fact that the client did not previously perform an explicit "stop recording" action. Had the FCM been released with `neutral` set to `False`, the recording should continue.

Either all three methods `Reserve`, `Release`, and `GetReservationStatus`, and the events `ReserveIndication` and `ReleaseIndication` shall be implemented by the FCM, or none of them shall be. If these methods are not supported, the error code `ENOT_IMPLEMENTED` is returned if any of them is invoked.

The execution of this API does not depend on the power state of the FCM. This means that a FCM shall not use the error code of `ESTANDBY`.

Error codes

- `Fcm::ENO_RELEASE` – there was no reservation to be released for this client
- `ENOT_IMPLEMENTED` – the resource does not allow reservations

Fcm::GetReservationStatus

Prototype

```
Status Fcm::GetReservationStatus(
   out boolean primaryPossible,
   out boolean secondaryPossible,
   out sequence<ClientRecord> clientRecords)
```

Parameters

- `primaryPossible` – only if `True`, the resource can accept a non-intrusive primary reservation
- `secondaryPossible` – only if `True`, the resource can accept a (non-intrusive) secondary reservation
- `clientRecords` – the list of primary and secondary clients of this resource; if the list is empty, there are no clients; if `primaryPossible` is `False`, the first record denotes the primary client, otherwise it denotes a secondary client. The safe parameter size limit is 10

`ClientRecord` values.

Description

A Resource Manager can use this method to retrieve reservation-related data from the FCM. Any other software element may also use this method.

Either all three methods `Reserve`, `Release`, and `GetReservationStatus`, and the events `ReserveIndication` and `ReleaseIndication` shall be implemented by the FCM, or none of them shall be. If these methods are not supported, the error code `ENOT_IMPLEMENTED` is returned if any of them is invoked.

Error codes

- `ENOT_IMPLEMENTED` – the resource does not allow status retrieval

Fcm::GetWorstCaseStartupTime

Prototype

```
Status Fcm::GetWorstCaseStartupTime(out long seconds)
```

Parameters

- `seconds` – worst case number of seconds needed to start a command

Description

Queries for the longest time an FCM takes to start one of its commands. It is a "worst case" time in the sense that it reflects the longest time the slowest FCM command requires to become activated.

This may be used for a Scheduled Action start time setup, for synchronizing the start-up of all devices in the Scheduled Action. It is up to the application to get worst case startup times of FCMs before setting a Scheduled Action.

Error codes

- `ENOT_IMPLEMENTED` – no worst case startup time available

Fcm::SetPlugSharing

Prototype

```
Status Fcm::SetPlugSharing(
      in ushort plugNum,
      in boolean canShare)
```

Parameters

- `plugNum` – an output FCM plug
- `canShare` – desired sharing state for the plug

Description

`SetPlugSharing` is intended for use in parental control and other situations where one application prevents other applications from making connections involving a particular source plug. If `canShare` is `False`, then only the "plug owner" (the software element which disabled sharing) will be allowed to make connections using the plug (existing connections using the plug and established by other software elements will be dropped by Stream Managers).

Note that FCM plug sharing and FCM reservation are orthogonal – in particular, reserving an FCM does not necessarily prevent other applications from connecting to its plugs.

A change in the plug sharing state causes a `PlugSharingChanged` event to be posted.

It is recommended that an application call `SetPlugSharing` after establishing connection (using the source plug) rather than before.

Error codes

- `Fcm::ENO_ADDR` – if the specified output plug does not exist for this FCM

Fcm::IecAttach

Prototype

```
Status Fcm::IecAttach(
      in IecPlug pcr,
      in InternalPlug plug)
```

Parameters

- `pcr` – the plug control register (PCR) to attach to the virtual FCM.
- `plug` – the FCM plug to attach to the PCR

Description

`Fcm::IecAttach` will be called by an FAV DCM when the DCM establishes a connection to a virtual FCM (i.e., `Dcm::Connect` is invoked). In the case of connections involving virtual FCMs, the DCM will only allow attachments between IEC 61883 plugs and the virtual FCM. This restriction may be relaxed in future versions of HAVi.

As a result of this call, the target (virtual) FCM can now start consuming the isochronous stream from the specified PCR (`pcr` is an iPCR and `plug` is an input FCM plug) or producing the stream (`pcr` is an oPCR and `plug` is an output FCM plug).

Error codes

- `ENOT_IMPLEMENTED` – if this FCM is not a virtual FCM
- `Fcm::ENO_ADDR` – the `pcr` or `plug` values are invalid
- `Fcm::EATTACH` – if the attachment cannot be made (e.g., direction mismatch or a sink already has an attachment)

Fcm::IecDetach

Prototype

```
Status Fcm::IecDetach(
      in IecPlug pcr,
      in InternalPlug plug)
```

Parameters

- `pcr` – the plug control register (PCR) to detach from the virtual FCM.
- `plug` – the FCM plug to detach from the PCR

Description

`Fcm::IecDetach` will be called by a (FAV) DCM when the DCM breaks the connection to a

virtual FCM (i.e., `Dcm::Disconnect` is invoked). As a result of this call, the target (virtual) FCM should now stop consuming or producing the isochronous stream.

Error codes

- `ENOT_IMPLEMENTED` – if this FCM is not a virtual FCM
- `Fcm::ENO_ADDR` – the `pcr` or `plug` values are invalid
- `Fcm::EATTACH` – if the attachment does not exist

5.7.4 Functional Component Module Events

PowerStateChanged

Prototype

```
void PowerStateChanged(in boolean powerState)
```

Parameters

- `powerState` the current power state of the functional component represented by the FCM

Description

Posted when there is a change of the power state of the functional component represented by the FCM.

PowerFailureImminent

Prototype

```
void PowerFailureImminent()
```

Description

Indication that a power failure is imminent. This event indicates that the part of the device represented by the FCM is about to enter an unpowered state. The consequences of entering the unpowered state are implementation specific. This event shall only be posted if at least one of the other FCMs of the DCM and the DCM itself remain powered, otherwise `Dcm::PowerFailureImminent` shall be posted.

ReserveIndication

Prototype

```
void ReserveIndication(in SEID client,
   in boolean primary)
```

Parameters

- `client` – the client for which the FCM was reserved
- `primary` – `True` in case of a reservation with primary access rights, `False` in case of a reservation with secondary access rights.

Description

This is an indication of the reservation of the posting FCM. If it is preempted by the client, both a `ReleaseIndication` and `ReserveIndication` are posted by the FCM. If a reservation on behalf of a client changes its access rights (from secondary to primary, or the other way around),

also both a `ReleaseIndication` and `ReserveIndication` are posted by the FCM.

Clients should subscribe to these events because they want to track the reservation status of FCMs that they have reserved (other clients can preempt their resources).

Either all three methods `Reserve`, `Release`, and `GetReservationStatus`, and the events `ReserveIndication` and `ReleaseIndication` shall be implemented by the FCM, or none of them shall be. If these methods are not supported, the error code `ENOT_IMPLEMENTED` is returned if any of them is invoked.

ReleaseIndication

Prototype

```
void ReleaseIndication(in SEID client,
   in boolean primary, in boolean neutral)
```

Parameters

- `client` – the client for which the FCM was released
- `primary` – `True` in case of a release with primary access rights, `False` in case of a release with secondary access rights.
- `neutral` – `True` in case a release of the FCM resulted in its neutral state, `False` in case of a release without a state change of the FCM.

Description

This is an indication of the release of the posting FCM, either explicitly by the client, or implicitly (by preemption on behalf of another client, or by disappearance of the client).

Either all three methods `Reserve`, `Release`, and `GetReservationStatus`, and the events `ReserveIndication` and `ReleaseIndication` shall be implemented by the FCM, or none of them shall be. If these methods are not supported, the error code `ENOT_IMPLEMENTED` is returned if any of them is invoked.

PlugSharingChanged

Prototype

```
void PlugSharingChanged(
      in ushort plugNum
      in boolean canShare,
      in SEID owner)
```

Parameters

- `plugNum` – an output FCM plug
- `canShare` – the current sharing state of the specified plug
- `owner` – the SEID of the software element which disabled sharing

Description

Posted when the sharing state of a plug changes. This happens either:

- as a result of an invocation of `Fcm::SetPlugSharing` that alters the plug sharing state
- if sharing is disabled and the owner disappears (in which case `canShare` shall be `True`)

If `canShare` is `True` the value of `owner` is undefined and should be ignored.

5.8 Device Control Module Manager

5.8.1 Services Provided

Service	Comm Type	Locality	Access
DcmManager::SetPreference	M	global	trusted
DcmManager::GetPreference	M	global	all
DcmManager::GetDeviceIcon	M	global	all
DcmManager::InstallDcm	M	global	trusted
DcmManager::UninstallDcm	M	global	trusted
DcmManager::DMInitialization	M	global	DCM Managers
DcmManager::DMInitialInquiry	M	global	DCM Managers
DcmManager::DMInquiry	M	global	DCM Managers
DcmManager::DMCommand	M	global	DCM Managers
DcmManager::DMGetDcm	M	global	DCM Managers
DcmInstallIndication	E	global	all
DcmUninstallIndication	E	global	all

The entries starting with the prefix `DM` are DCM management protocol messages.

5.8.2 DCM Manager Data Structures

VMID

Definition

```
typedef octet ModelId[3];

struct VMID {
   VendorId      deviceVendor;
   ModelId       deviceModel;
};
```

Description

`VendorId` is the **Vendor_ID**, and `ModelId` is the **Model_ID** representing a BAV device model, as specified in the SDD data. A `VMID` (Vendor Model Identifier) therefore specifies a device model from some vendor.

GuestId

Definition

```
union GuestId switch(boolean) {
      case True:    VMID group;
      case False:   GUID single;
};
```

Description

Specification of a guest designation. `GUID` denotes a single guest, while `VMID` denotes a group of guests.

PreferenceId

Definition

```
enum PreferenceId {
   DCM_PREFER_VENDOR_HOST,
   DCM_PREFERRED_HOST,
   DCM_PREFERRED_URL,
   DM_PREFERRED_URL_DEVICE
};
```

Description

The set of DCM management preference identifiers.

PreferenceValue

Definition

```
union PreferenceValue switch(PreferenceId) {
      case DCM_PREFER_VENDOR_HOST:  boolean preferVendorHost;
      case DCM_PREFERRED_HOST:      GUID    preferredHost;
      case DCM_PREFERRED_URL:       URLString  preferredURL;
      case DM_PREFERRED_URL_DEVICE: GUID    preferredURLDevice;
};
```

Description

Specification of a preference value.

ProfileRecord

Definition

```
struct ProfileBody {
   long transferredDcmCodeUnitSize;
   long installedDcmCodeSpace;
   long installedDcmWorkingSpace;
   Version messageVersion;
};

union ProfileRecord switch(boolean) {
      case True:    ProfileBody   Profile;
      case False:   ; //empty
};
```

Description

Specification of a DCM code unit profile, in accordance to section 9.10.7.

URLString

Definition

```
typedef sequence<octet> URLString;
```

Description

Specification of a URL string. The length shall not exceed 256 bytes. It is not null-terminated (see section 9.10.8).

DMCommandType

```
enum DMCommandType {
      LEADER, INSTALL_INV, UNINSTALL_INV,
      INSTALL_URL_PREF, INSTALL_URL_BAV, INSTALL_CODE_BAV,
      INSTALL, UNINSTALL, INSTALLED, NOT_INSTALLED
};
```

DMCommandResult

```
enum DMCommandResult { ACCEPTED, REJECTED, BUSY };
```

DMGetDcmResult

```
enum DMGetDcmResult {
      CONTINUED, FINISHED,
      FAILED_SCHEME, FAILED_CODE, FAILED_ACCESS, FAILED
};
```

DcmInstallResult

```
enum DcmInstallResult {
      IGNORED,
      URL_PREF, URL_BAV,
      CODE_BAV, PROP_VENDOR, PROP,
      FAILED
};
```

DcmInstallConflict

```
enum DcmInstallConflict {
      NONE,
      VENDOR, HOST, URL, VENDOR_HOST,
      VENDOR_URL, HOST_URL, VENDOR_HOST_URL
};
```

DcmUninstallResult

```
enum DcmUninstallResult { IGNORED, COMMANDED, SPONTANEOUS };
```

5.8.3 DCM Manager API

Upon a network reset event, the DCM management system shall perform installation and uninstallation activities autonomously, according to the description in section 3.6.1. This section describes the API services of DCM Managers. Please refer to section 5.8.5 for the overall protocol description and the associated concepts.

DcmManager::SetPreference

Prototype

```
Status DcmManager::SetPreference(
      in GuestId guest,
      in boolean set,
      in PreferenceValue value)
```

Parameters

- `guest` – the guest or guest model for which this preference is set (not interpreted for `DM_PREFERRED_URL_DEVICE`)
- `set` – if `True`, the preference is set; otherwise it is removed, and becomes unspecified (in which case `value` is not interpreted, except for `PreferenceId`)
- `value` – the preference value to be set

Description

A preference value is set and persistently stored (if possible) on or through the host device running the DCM Manager that receives this message. However, a DCM Manager may refuse the setting of a preference if it does not have the capability. The removal of a preference always succeeds, even if the preference was not set.

Error codes

- `EINVALID_PARAMETER` – faulty preference value parameters were specified
- `DcmManager::EVOLATILE` – the value was set, but not persistently
- `DcmManager::EFAIL` – the value was not set because the guest parameter is known not to designate a BAV or LAV device or model.

DcmManager::GetPreference

Prototype

```
Status DcmManager::GetPreference(
      in PreferenceId preference,
      in GuestId guest,
      out boolean set,
      out PreferenceValue value)
```

Parameters

- `preference` – the identifier of the preference for which the value is retrieved
- `guest` – the guest or guest model for which this preference is retrieved (not to be interpreted for `DM_PREFERRED_URL_DEVICE`)
- `set` – if `True`, the preference was set, and `value` is valid; otherwise the preference is unspecified
- `value` – the retrieved preference value

Description

A preference value is retrieved from or through the host device on whose DCM Manager the method is invoked. If the preference is not found `value` shall not contain correct data.

Error codes

- `EINVALID_PARAMETER` – a faulty preference parameter was specified
- `DcmManager::EFAIL` – a value was not retrieved because the guest parameter is known not to designate a BAV or LAV device or model.

DcmManager::GetDeviceIcon

Prototype

```
Status DcmManager::GetDeviceIcon(
      in GUID guest,
      out DeviceIcon icon)
```

Parameters

- `guest` – GUID of a BAV or LAV device for which a device icon is requested
- `icon` – a representation for a BAV or LAV device. (The `DeviceIcon` type is discussed in section 5.6.2.)

Description

An icon is retrieved for `guest`. This method makes it possible to display a graphical or textual representation of a guest for which no DCM code unit is installed, e.g., to show guests for which DCM code unit installations have failed. For a BAV device, a representation is coded in the SDD data. How a representation for an LAV device may be retrieved is not specified. If no representation can be retrieved, `DcmManager::EFAIL` will be returned and the value of `icon` is undetermined.

The device icon shall be constructed from the **HAVi_User_Preferred_Name** field in the SDD of the device and from the **HAVi_Device_Icon_Bitmap** field if it is available.

Error codes

- `DcmManager::EFAIL` – a device icon could not be retrieved (possibly invalid guest specified)

DcmManager::InstallDcm

Prototype

```
Status DcmManager::InstallDcm(in GUID guest)
```

Parameter

- `guest` – GUID of a guest for which a DCM code unit shall be installed

Description

A DCM code unit shall be installed for a BAV or LAV device for which no DCM code unit is currently installed, if possible. If the method returns successfully, the installation is in progress, and a `DcmInstallIndication` event shall designate the result of the installation attempt. The preferences specified for the guest shall determine the installation results, in the same way as for the autonomous installation procedure performed by the DCM management system.

A final follower that receives an `InstallDcm` message sends a `DMCommand` (with `command` value `INSTALL_INV`) to the final leader. If no final leader is known, the method shall return with error code `EFAIL`. The final leader shall accept the command immediately, and `InstallDcm` shall return successfully. (The final leader handles the command after accepting it.) Only if the final follower knows the specified guest parameter is not valid, `DMCommand` shall not be sent to the final leader, and the final follower shall immediately return with error code `EFAIL`. This is the case if the specified guest is not in the network or does not correspond to a BAV or LAV device.

Note: If the DCM code unit is already installed locally, the receiving final follower may not handle (reject) the request locally, but should still involve the final leader.

Error codes

- `DcmManager::EFAIL` – the installation failed (possibly invalid guest specified)

DcmManager::UninstallDcm

Prototype

```
Status DcmManager::UninstallDcm(in GUID guest)
```

Parameter

- `guest` – GUID of a guest for which the DCM code unit shall be uninstalled

Description

A DCM code unit, if there is any, shall be uninstalled for `guest`. After the method returns successfully, the uninstallation is in progress, and a `DcmUninstallIndication` event shall designate if a DCM code unit was uninstalled or not.

A final follower that receives an `UninstallDcm` message sends a `DMCommand` (with `command` value `UNINSTALL_INV`) to the final leader. If no final leader is known, the method shall return with error code `EFAIL`. The final leader shall accept the command immediately, and `UninstallDcm` shall return successfully. (The final leader handles the command after accepting it.) Only if the final follower knows the specified guest parameter is not valid, `DMCommand` shall not be sent to the final leader, and the final follower shall immediately return with error code `EFAIL`. This is the case if the specified guest is not in the network or does not correspond to a BAV or LAV device.

Note: If the DCM code unit is installed locally, the receiving final follower need not involve the final leader, and may uninstall the code unit directly.

Error codes

- `DcmManager::EFAIL` – the uninstallation failed (possibly invalid guest specified)

DcmManager::DMInitialization

Prototype

```
Status DcmManager::DMInitialization(
      out boolean URLDeviceSet,
      out GUID URLDevice)
```

Parameters

- `URLDeviceSet` – `True` if the value of `URLDevice` is valid, `False` otherwise
- `URLDevice` – the value of preference `DM_PREFERRED_URL_DEVICE` set at the receiving

initial follower (any value if the preference is not set)

Description

This is a protocol message (see section 5.8.5). This message is sent from the initial leader to all initial followers after the initial leader has received a `NetworkReset` event or is powered up. If a DCM Manager receives this message, it shall verify that it originates from the current initial leader (through the use of `Cmm1394::GetGuidList`). If not, it shall return with error code `EFAIL`. If the current final leader receives this message, it shall stop acting as a final leader (a new one shall be selected subsequently).

Error codes

- `DcmManager::EFAIL` – the message was not received from the assumed initial leader

DcmManager::DMInitialInquiry

Prototype

```
Status DcmManager::DMInitialInquiry(
      out sequence<GUID> guestList)
```

Parameter

- `guestList` – the GUIDs of the guests for which a DCM code unit is locally installed. The safe parameter size limit of this list is bounded by the safe parameter size limit of the lists returned by `Cmm1394::GetGuidList`.

Description

This is a protocol message (see section 5.8.5). It is sent from the final leader to a final follower to find out for which guests the final follower has installed a DCM code unit. The final leader shall only take additional autonomous actions (`DMInquiry`, `DMCommand`) for guests for which no DCM code unit is installed on any host (installation), or for guests for which more than one DCM code unit is found to be installed (uninstallation).

It shall be the first method used by the final leader after it is selected, and shall be sent to all final followers, even if there are no guests in the network. From this message, a final follower learns which DCM Manager is the final leader.

DcmManager::DMInquiry

Prototype

```
Status DcmManager::DMInquiry(
      in GUID guest,
      in ProfileRecord profile,
      out boolean preferVendorHost,
      out boolean preferVendorHostForModel,
      out GUID preferredHost,
      out GUID preferredHostForModel,
      out URLString preferredURL,
      out URLString preferredURLForModel,
      out boolean sameVendorHost,
      out boolean acceptable,
      out boolean installed,
      out short dcmCount)
```

Parameters

- guest – the guest for which an (un)installation inquiry is performed
- profile – the profile that the final follower shall interpret to determine if it can install the corresponding DCM Java code unit (empty if it does not apply in the inquiry)
- preferVendorHost – the value of preference DCM_PREFER_VENDOR_HOST set at the final follower (False if the preference is not set)
- preferVendorHostForModel – the preferVendorHost preference for the VMID of the guest (typically False if the preference is unspecified)
- preferredHost – the value of preference DCM_PREFERRED_HOST set at the final follower (the guest's GUID if the preference is not set)
- preferredHostForModel – the preferredHost preference for the VMID of the guest (the guest's GUID if the preference is not set)
- preferredURL – the value of preference DCM_PREFERRED_URL set at the final follower (an empty string if the preference is not set)
- preferredURLForModel – the preferredURL preference for the VMID of the guest (an empty string if the preference is not set)
- sameVendorHost – True only if the final follower identifies the guest to be from the same vendor as itself
- acceptable – (1) if profile is empty: True only if the host is able to install a proprietary DCM code unit for the guest; (2) if profile is not empty: True only if the host is able to upload and install the DCM Java code unit for the guest whose characteristics are represented by profile. Note that if the profile contains a version number higher than the version number of the host, acceptable should be False.
- installed – True only if a DCM code unit for the guest is already installed
- dcmCount – the number of DCM code units currently installed on the host

Description

This is a protocol message (see section 5.8.5). This message is sent from the final leader to a final follower. The final leader requests for some guest data locally available at the final follower. The final leader sends a non-empty profile for an uploadable DCM Java code unit only if the (FAV) final follower should reply whether it is able to install the corresponding code unit. (This decision shall be based on the parameters contained in the profile.)

Each DCM Manager with preference settings for VMIDs (guest models) shall verify for a BAV guest whether the VMID applies to it. If so, one or more of the preferences that apply to the guest model shall also be returned. Note that a DCM Manager can consult the BAV device's SDD to acquire the necessary model ID data.

This message may be invoked by any remote DCM Manager at any time, to acquire installation data for a guest.

DcmManager::DMCommand

Prototype

```
Status DcmManager::DMCommand(
      in DMCommandType command,
      in GUID device,
      in URLString preferredURL,
      out DMCommandResult result)
```

Parameters

- command:

LEADER – selected for final leadership
INSTALL_INV – install invocation
UNINSTALL_INV – uninstall invocation
INSTALL_URL_PREF – install the Java code unit designated by the preferred URL (only then the preferredURL parameter shall not be the empty string)
INSTALL_URL_BAV – install the Java code unit designated by the URL contained in the BAV device
INSTALL_CODE_BAV – install the Java code unit contained in BAV device
INSTALL – install in a proprietary manner
UNINSTALL – uninstall
INSTALLED – installation succeeded
NOT_INSTALLED – installation failed

- device – a host or guest value, depending on the command type
- preferredURL – the URL that a (FAV) device shall use to install a DCM code unit
- result:

ACCEPTED – accepted (always for command values LEADER, INSTALL_INV, UNINSTALL_INV, UNINSTALL, INSTALLED, NOT_INSTALLED; possible for command values INSTALL_URL_PREF, INSTALL_URL_BAV, INSTALL_CODE_BAV, INSTALL)
REJECTED – rejected (only possible for command values INSTALL_URL_PREF, INSTALL_URL_BAV, INSTALL_CODE_BAV, INSTALL)
BUSY – busy (only possible for command values INSTALL_URL_PREF, INSTALL_URL_BAV, INSTALL_CODE_BAV, INSTALL)

Description

This is a protocol message (see section 5.8.5). The specific cases are :

- LEADER – The message is sent by the initial leader to an initial follower to declare the follower as the final leader. It shall subsequently behave accordingly. Parameter device indicates a URL capable device, unless it is the receiver's GUID (the receiver knows whether it supports URL access). The receiver shall ignore this command and return with error code EFAIL if it does not originate from the initial leader.

- INSTALL_INV, UNINSTALL_INV – The message is sent by a final follower to the final leader upon an invocation of DcmManager::InstallDcm or DcmManager::UninstallDcm by a software element on the final follower. Parameter device indicates the guest to which the request applies. If the receiver assumes it is not the final leader, an EFAIL error code shall be returned by it and the command is ignored.

- INSTALL_URL_PREF, INSTALL_URL_BAV, INSTALL_CODE_BAV, INSTALL – The message is sent by the final leader to a final follower. Parameter device indicates the guest for which a DCM code unit should be installed. A command for installing a DCM code unit shall be accepted by a final follower if it expects to be able to install it. Only if it has insufficient resources, an install command shall be rejected. However, it may be that the specific installation requested cannot *currently* be performed by the final follower, because it is installing another DCM code unit. In this temporary condition, it shall return a busy result. A final follower that accepted an installation command shall normally return a DMCommand to the final leader to inform it about the installation attempt (see below). If the sender is assumed not to be the final leader, an EFAIL error code shall be returned by the receiver.

- `UNINSTALL` – The message is sent by the final leader to a final follower. Parameter `device` indicates the guest for which a DCM code unit should be uninstalled. A command for uninstalling a DCM code unit shall always be accepted and immediately executed by a final follower, although it will take no action if there is no installed code unit (it shall not post a `DcmUninstallIndication` event). If the sender is assumed not to be the final leader, an `EFAIL` error code shall be returned by the receiver.

- `INSTALLED, NOT_INSTALLED` – The message is sent by a final follower to the final leader to inform it about the result of an accepted installation request. Parameter `device` indicates the guest for which the installation was attempted. These messages shall be voided if the final leader is unknown. If the receiver assumes it is not the final leader, an `EFAIL` error code shall be returned by it.

Error codes

- `DcmManager::EFAIL` – the message was not received from the assumed initial/final leader

DcmManager::DMGetDcm

Prototype

```
Status DcmManager::DMGetDcm(
      in URLString URL,
      in long demandBegin,
      in long demandEnd,
      out long supplyBegin,
      out long supplyEnd,
      out sequence<octet> byteArray,
      out DMGetDcmResult result)
```

Parameters

- `URL` – the locator of the DCM Java code unit to be uploaded
- `demandBegin` – the number of the first byte of the array wanted (1 indicates the first byte of the code unit; 0 indicates that the sender is no longer interested in the code unit)
- `demandEnd` – the number of the last byte of the array wanted (may be larger than the last byte of the code unit)
- `supplyBegin` – the number of the first byte of the byte array delivered (corresponding to the first byte of byteArray; 0 if it is empty)
- `supplyEnd` – the number of the last byte of the byte array delivered (corresponding to the last byte of byteArray)
- `byteArray` – the array of bytes that are a part of the code unit being uploaded. The safe parameter size limit is 512 bytes (the 1394 asynchronous packet size for bus speeds of 100 Mbps).
- `result`:
  `CONTINUED` – successful retrieval, more bytes to follow
  `FINISHED` – successful retrieval, no more bytes to follow
  `FAILED_SCHEME` – unsuccessful retrieval, URL scheme could not be handled
  `FAILED_CODE` – unsuccessful retrieval, code unit was not found
  `FAILED_ACCESS` – unsuccessful retrieval, URL access was aborted
  `FAILED` – unsuccessful retrieval (other cause)

Description

This message is sent from a final follower to a DCM manager that is believed to be able to access the DCM Java code unit at the specified URL location (typically the final leader). The URL value

is either the one presented by the final leader as the `preferredURL` parameter in `DMCommand`, or it is the one found in the BAV device (depending on the leader command). The URL shall be extended with the HAVi DCM code extension, if none is specified.

The requester shall attempt to retrieve and install the code unit by sending this message one or more times. Unless the final follower already has the code unit, the message shall be sent to the final leader first. If the final leader fails to retrieve a code unit, the final follower is free to try any DCM Manager or URL access capable FCM, or use proprietary means to acquire the code unit. Trying the final leader first allows use of cached code units at the final leader device, such that, for example, BAV devices of the same vendor and model may amortize a URL access effort for their DCM code unit.

The number of bytes requested may correspond to the maximum load the requester can accept; it may be larger than the number of bytes actually retrieved. The code unit sender will always indicate which part of the code unit was delivered, and whether it was the last part. Note that the actual number of bytes delivered by the code unit sender may be bound by the amount of memory working space it has.

It depends on the URL scheme and the storage medium how long it will take to fetch a number of bytes. It is allowed, particularly if code unit retrieval takes a long time, for the replied array of bytes to be smaller than can be handled by receiver or by the final follower. At any time, the final follower can decide whether to continue requesting additional bytes, or to stop the process if it takes too long. As a courtesy, the final follower shall send a final `DMGetDcm` message with `demandBegin` set to 0, to indicate to the code unit sender that it can stop retrieving data on its behalf.

5.8.4 DCM Manager Events

DcmInstallIndication

Prototype

```
void DcmInstallIndication(
      in GUID guest,
      in DcmInstallResult result,
      in DcmInstallConflict conflict)
```

Parameters

- `guest` – GUID of a guest device for which a DCM code unit installation result is posted
- `result`:
  `IGNORED` – already installed (possible only upon an install invocation)
  `URL_PREF` – Java code unit installed from preferred URL location
  `URL_BAV` – Java code unit installed from URL location obtained from BAV device
  `CODE_BAV` – Java code unit obtained from BAV device installed
  `PROP_VENDOR` – proprietary code unit installed by a host from the same vendor as the guest
  `PROP` – proprietary code unit installed by a host not from the same vendor as the guest
  `FAILED` – installation in network failed
- `conflict`:
  `NONE` – no conflict
  `VENDOR` – prefer vendor host conflict
  `HOST` – preferred host conflict
  `URL` – preferred URL conflict
  `VENDOR_HOST` – prefer vendor host conflict and preferred host conflict

VENDOR_URL – prefer vendor host conflict and preferred URL conflict
HOST_URL – preferred host conflict and preferred URL conflict
VENDOR_HOST_URL – prefer vendor host conflict and preferred host conflict and preferred URL conflict

Description

This is an indication of an installation result for a guest. It is always posted by the final leader after a successful installation, or after no installation in the network is found to be possible. A reported conflict can relate to either guest and/or guest model preferences.

Note that the DCM management protocol does not use this event. It is intended for (system) applications.

DcmUninstallIndication

Prototype

```
void DcmUninstallIndication(
      in GUID guest,
      in DcmUninstallResult result)
```

Parameters

- guest – GUID of a guest device for which a DCM code unit uninstallation result is posted
- result:
  IGNORED – uninstallation failed (upon an uninstall invocation if there is no DCM code unit)
  COMMANDED – uninstallation succeeded (not on initiative by DCM code unit)
  SPONTANEOUS – uninstallation succeeded (on initiative by DCM code unit)

Description

This is an indication of an uninstallation attempt for a guest. The specific cases are:

- IGNORED – If the final leader is requested to uninstall a DCM code unit, but finds none in the network, it posts the event with this result. (Note that it is possible there are guests in the network for which no DCM code unit is installed.)

- COMMANDED – If a final follower uninstalls a DCM code unit upon request of the final leader or another software element, it posts the event with this result.

- SPONTANEOUS – If a DCM code unit is uninstalled by invoking uninstalled() on the DCM Manager on the code unit's own initiative, the local DCM Manager posts the event with this result.

Note that the DCM management protocol does not use this event. It is intended for (system) applications.

5.8.5 DCM Management Protocol

The DCM management system is constructed from a distributed group of DCM Managers on FAV and IAV devices. DCM Managers interact on a peer-to-peer basis to implement the DCM management service, while in turn using services of local software elements. These are the CMM, Messaging System, Event Manager, Registry, and DCM code units. Each DCM Manager can read SDD data directly from devices connected to the HAVi 1394 network. The DCM management

protocol supports the use of device storage and Internet access facilities.

In a nutshell, each DCM Manager starts with a leader election after a network reset event is received. One DCM Manager will be selected as the final *leader*. All DCM Managers are final *followers*, and subordinate to the final leader. Note that the final leader also plays the role of a final follower in the protocol description. Final leader and followers subsequently collaborate to install DCM code units autonomously for each guest found on the network (for which a DCM code unit is not yet installed). In addition to automatic DCM management, each DCM Manager may also accept method invocations by software elements. The final leader will control most of the protocol activities.

5.8.5.1 Leader Election

After a device is powered up, or after a `NetworkReset` event is received, a DCM Manager behaves as follows:

- The GUID list is retrieved through `Cmm1394::GetGuidList`. The relevant SDD data of all devices are retrieved: **HAVi_Device_Class**, **HAVi_DCM_Manager**, **Model_ID**, **Vendor_ID**, **EUI-64** (GUID). Devices without such SDD data are classified as LAV devices.

- An IAV device without a DCM Manager (derived from **HAVi_DCM_Manager**) shall be ignored, and shall not be a host for any guest on the HAVi 1394 network.

- From all host GUIDs, the highest bit order-reversed GUID is calculated, and the DCM Manager on the device with this GUID is declared the *initial* leader. The reversal prevents devices from certain vendors acting as the initial leader in many network configurations (since the GUID starts with a vendor identifier.) Note that all devices read the same GUID list, and will declare the same DCM Manager as initial leader. All other DCM Managers are initial followers.

At this time, each DCM Manager knows if it is the initial leader or an initial follower. Each device knows which other DCM Managers there are, and which SEIDs they have. (The SEID is the concatenation of the device GUID and the fixed DCM Manager software element handle.) Message passing between DCM Managers is enabled. Each DCM Manager shall be registered.

The initial leader selects a final leader, and possibly a URL access capable host or guest in the following manner:

- To all initial followers the initial leader sends a `DMInitialization` message, and awaits all responses. The responses may yield one or more `DM_PREFERRED_URL_DEVICE` preferences (declaring URL access capabilities) for FAV, IAV, BAV, and/or LAV devices.

- The selection of the final leader is as follows:
  - If there are FAV devices with a declared URL access capability, one of them is selected in a proprietary way.
  - Otherwise, if there are IAV devices with a declared URL access capability, one of them is selected in a proprietary way.
  - Otherwise, if there are FAV devices, one of them is selected in a proprietary way.
  - Otherwise, if there are IAV devices, one of them is selected in a proprietary way.

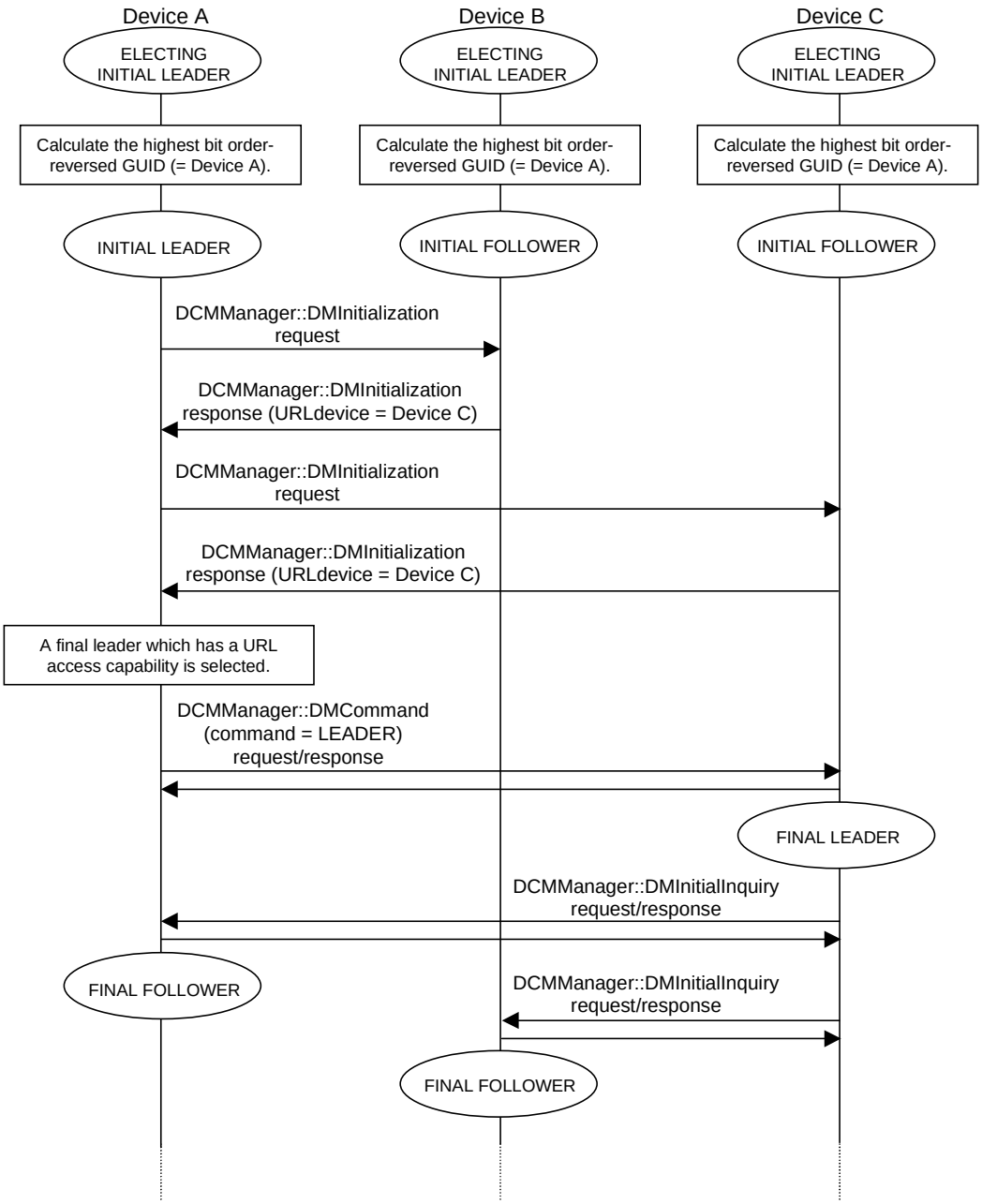

Figure 32. DCM Manager Protocol

- If there is no existing FAV or IAV device with a declared URL access capability, the initial leader verifies whether at least one of the `DM_PREFERRED_URL_DEVICE` preferences corresponds to an existing BAV or LAV device. If so, it selects any of them.

- A `DMCommand` is sent by the initial leader to the selected DCM Manager to announce to the receiver it is the final leader. A URL access capable guest GUID may also be specified (unless the receiver's GUID is provided). If the final leader has URL access capability itself, it may ignore the guest GUID.

The final leader starts by sending DMInitialInquiry messages to all final followers, so each DCM Manager will know which one is the final leader. The autonomous operation phase then commences.

The process sketched in this section is represented in the message sequence chart for devices A, B, and C. Note that DMInitialization requests may be sent in parallel to increase the performance of the election process.

5.8.5.2 Autonomous Operation

The final leader will take the initiative to handle DCM code unit installations. The final followers will wait for and handle DMInquiry and DMCommand messages from the final leader. The final leader will control the operation as follows, for each guest in the network for which no DCM code unit is already installed (as reported by DMInitialInquiry):

- To each final follower an initial DMInquiry message is sent with an empty profile parameter. From all received inquiry records, the final leader learns the following:
  - The preferences set for the guest and guest model, if any.
  - For each host, whether the guest is from the same vendor, and whether it can install a proprietary DCM code unit for the guest.
  - For each host, whether a DCM code unit is already installed for the guest.
  - For each host, the number of installed DCM code units.
- If no installed DCM code unit was found for the guest, the final leader shall send additional DMInquiry and DMCommand messages to all or some final followers, and proceed according to the rules in sections 3.6.1 and 3.6.2 to install a DCM code unit. Additional DMInquiry messages must carry the profile of the DCM code unit to be installed. The profile is empty in the case of a proprietary DCM code unit. Note that the final leader must acquire a URL-specified profile before a DCM Manager can install the corresponding code unit.

For each install and uninstall action, initial DMInquiry messages will be sent to all final followers. The final leader shall not rely on its local installation knowledge. (Note that DCM code units can be uninstalled spontaneously.) The final leader shall not attempt to install another code unit for a guest for which the code unit has been uninstalled. Only after a network reset or after an invoked installation request (DcmManager::InstallDcm) shall such an attempt be made.

If the final leader finds more than one installed DCM code unit for a guest, it shall uninstall all but one.

If the final leader has no URL capability itself, it shall employ the selected URL capable guest, if available. If no DCM code unit for it is already installed, the final leader shall attempt to install a DCM code unit for it, in the same way as for any other guest. Subsequent URL access can take place through the DCM or one or more FCMs installed for the guest.

If an install command results in a “busy” indication from a candidate host, the final leader may either choose an equally suited host, or it may first try to install a DCM code unit for another guest. However, the final leader shall not try to install a DCM code unit on a less suited host if there is at least one better suited host with a “busy” indication. This indication shall be temporary, and shall only be used by a host if it is currently installing another DCM code unit. If all best suited hosts are busy, the final leader shall delay DCM code unit installation for the guest. It is recommended to retry busy candidate hosts not more often than once every three seconds.

Each final follower takes the initiative for the uninstallation of local DCM code units, without

involvement of the final leader. If a guest disappears from the network for which a DCM code unit is installed, the local final follower shall learn this upon a `NetworkReset` event. Note that a DCM code unit can also uninstall itself. In either case, an installation of another code unit is only attempted upon a `NetworkReset` event or an invoked installation request.

5.8.5.3 Protocol Details

The previous sections described the functional activities performed by a DCM Manager. In this section, a state machine is given that specifies how a DCM Manager should react to incoming events and messages from the system. Although the implementation of the required behavior is proprietary, the state machine can also serve as an implementation guideline.

Figure 33 shows a state machine of a DCM Manager, including major states and transitions in the protocol. The blocks represent states and the arrows represent transitions. A transition denotes a trigger, possibly with a condition and an associated action. Alternative transitions for an arrow are listed below each other. The ovals represent that the DCM Manager may reenter its previous *substate* (indicated by the dot) if one of the indicated transitions occurs. An arrow, on the other hand, indicates that the main state is (re)entered. The other symbols in the figure are explained in the next table.

T : C	a trigger T under the condition C
x → M+/-	message M was received from x and will be responded to with `SUCCESS`/`EFAIL`
x ← M+/-	message M was sent to x and responded to with `SUCCESS`/`EFAIL`
d	a remote DCM Manager
s	the DCM Manager to which the state transition diagram applies ("self")
i	the actual initial leader according to s (identified by `Cmm1394::GetGuidList` each time a trigger P or R occurs, and after a message I or L is received)
f	the final leader assumed by s
P	the device of s is powered up or activated
R	s receives a network reset event
I	a `DMInitialization` message
L	a `DMCommand` with command value `LEADER` (leader selection)
C	a `DMCommand` not with command values `LEADER`, `INSTALL_INV`, `UNINSTALL_INV`, `INSTALLED`, or `NOT_INSTALLED`

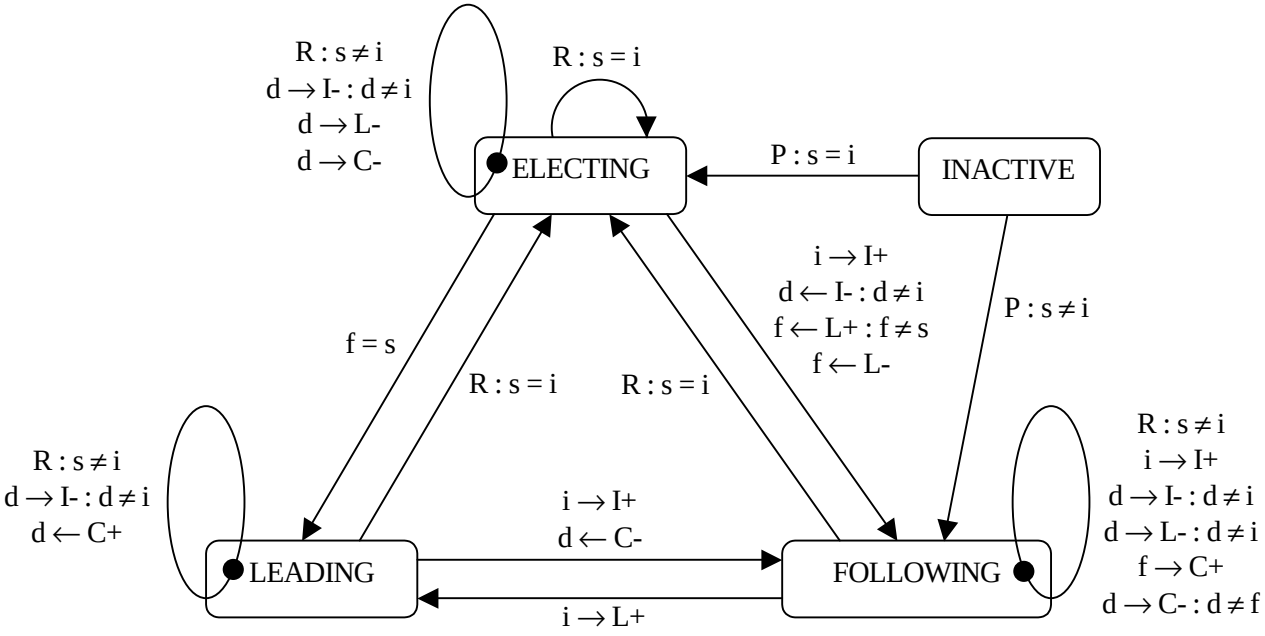

Figure 33. DCM Manager States

The main states of a DCM Manager are:

- ELECTING – As the initial leader it sends `DMInitialization` messages to all other DCM Managers, and awaits their responses. If it completes this activity, it selects either itself for final leadership and becomes LEADING, or it selects another DCM Manager for final leadership by sending it a message `DMCommand` for leader selection, and becomes FOLLOWING. It will also become FOLLOWING if another DCM Manager sends it a `DMInitialization` message and appears to be the real initial leader, and if it receives an `EFAIL` response on a `DMInitialization` or `DMCommand` message. If a network reset event occurs, it aborts handling the current `DMInitialization` messages, and restarts sending them to all actual DCM Managers.

- LEADING – As the final leader it starts by sending `DMInitialInquiry` messages to all DCM Managers. It will issue `DMInquiry` messages, and issue and accept `DMCommand` messages until it terminates the final leadership. It becomes FOLLOWING if it receives a `DMInitialization` message from the actual initial leader or an `EFAIL` response on a `DMCommand` message. If it becomes the initial leader, due to a network reset event, it becomes ELECTING.

- FOLLOWING – As an initial or final follower it awaits and handles incoming messages. If it receives a message `DMCommand` for leader selection from the actual initial leader it becomes LEADING. Other `DMCommand` messages will be rejected if they do not originate from the assumed final leader. If it becomes the initial leader, due to a network reset event, it becomes ELECTING.

The protocol takes into consideration possible timing variations between DCM Managers in the network, and in the delivery of events and messages. In particular, network reset events in general have no effect if the receiving DCM Manager does not identify itself as the initial leader. Note that in some circumstances, a new leader election can be in progress while the previous final leader is still issuing commands.

Protocol messaging shall rely on the transaction id mechanism of the Messaging System to match requests and responses. Unexpected responses shall be discarded.

In addition to the behavior according to the state machine, there are additional rules a DCM Manager implementation should follow:

- A DCM Manager shall always accept and handle protocol messages. If a message is not expected or cannot be handled, the receiver shall return with `EFAIL`. This can only be the case for `InstallDcm`, `UninstallDcm`, `DMInitialization`, and `DMCommand`.

- An invocation of `InstallDcm` and `UninstallDcm` by an application may return with `SUCCESS`, yet fail due to a network reset. However, the subsequent autonomous DCM management will “assimilate” or annihilate these invocations. The application may use watch-ons on DCMs and may subscribe to the events `DcmInstallIndication` and `DcmUninstallIndication` to handle these situations.

- A DCM Manager should abort a DCM code unit installation if it receives a `NetworkReset` event or a `DMInitialization` message during the installation process, before invoking install on the code unit – see section 3.6.3. (It is recommended to handle pending `NetworkReset` events before invoking install.)

- A DCM Manager shall stop sending any protocol message (`DMInitialization`, `DMInitialInquiry`, `DMInquiry`, `DMCommand`, `DMGetDcm`) to any DCM Manager, when the following occurs:
  - It receives `EUNIDENTIFIED_FAILURE` on a protocol message from another DCM Manager
  - It receives a `MsgTimeout` or `MsgError` event with SEID parameter denoting a DCM Manager.

  If it receives a protocol message from another DCM Manager while it stops handling protocol operation, it returns `EUNIDENTIFIED_FAILURE`. It restarts handling protocol operation, when it receives a `NetworkReset` event or a `DMInitialization` message. Note that it can handle a message except a protocol message, while it stops handling protocol operation.

5.9 Stream Manager

5.9.1 Services Provided

Service	Comm Type	Locality	Access
StreamManager::FlowTo	M	local	all
StreamManager::SprayOut	M	local	all
StreamManager::TapIn	M	local	all
StreamManager::Drop	M	global	all
StreamManager::GetLocalConnectionMap	M	global	all
StreamManager::GetGlobalConnectionMap	M	global	all
StreamManager::GetConnection	M	global	all
StreamManager::GetStream	M	global	all
ConnectionAdded	E	global	Stream Manager (all)
ConnectionDropped	E	global	Stream Manager (all)
ConnectionChanged	E	global	Stream Manager (all)

5.9.2 Stream Manager Data Structures

Direction

```
enum Direction {IN, OUT};
```

OperationalStatus

```
enum OperationalStatus {
```

```
    NORMAL,
    FAILURE,
    UNKNOWN,
};
```

`OperationalStatus` describes whether a connection is operating or not, more specifically:

- `NORMAL` indicates that the Stream Manager assumes the connection is operational.
- `FAILURE` indicates that the Stream Manager assumes the connection is not operational.
- `UNKNOWN` indicates that the Stream Manager cannot determine whether the connection is operational or not.

When a Stream Manager creates a `FlowTo` connection, it assures the source and sink are compatible, the connection then has `NORMAL` operational status.

When a Stream Manager creates a `SprayOut` or `TapIn` connection, it does not assure source/sink compatibility, the connection then has `UNKNOWN` operational status.

A connection will also have `UNKNOWN` operational status if static bandwidth allocation has been set and the bandwidth specified in the DCM output `IecPlug` is less than the maximum bandwidth of the stream type of the source FCM plug.

FailureReason

```
enum FailureReason {
    STREAM_TYPE_MATCH_FAILURE,
    TRANSMISSION_FORMAT_MATCH_FAILURE,
    BANDWIDTH_FAILURE,
};
```

After a connection has been established, its operational status may change. In particular the connection may change from `NORMAL` or `UNKNOWN` to `FAILURE` status. The Stream Manager does not drop connections that have `FAILURE` for their operational status, however applications may choose to do so.

`FailureReason` enumerates the situations that cause a `FAILURE` operational status:

- `STREAM_TYPE_MATCH_FAILURE` occurs if the Stream Manager cannot match source and sink stream type ID when servicing a `StreamTypeChanged` event or a `DeviceConnectionChanged` event.
- `TRANSMISSION_FORMAT_MATCH_FAILURE` occurs if the Stream Manager cannot match source and sink when servicing a `TransmissionFormatChanged` event or a `DeviceConnectionChanged` event.
- `BANDWIDTH_FAILURE` occurs if the Stream Manager cannot match source and sink bandwidth capability when servicing a `BandwidthRequirementChanged` event or a `StreamTypeChanged` event or a `DeviceConnectionChanged` event.
  `BANDWIDTH_FAILURE` may also occur if the source has failed to allocate bandwidth, or the stream type has changed when static bandwidth allocation is set and insufficient bandwidth is allocated.

If the conditions causing failure no longer apply, the operational status changes from `FAILURE` to `UNKNOWN` or `NORMAL`.

ConnectionState

```
struct ConnectionState {
```

```
   OperationalStatus        status;
   sequence<FailureReason>  failures;
};
```

The current state of a connection. `failures` shall be empty if `status` is `NORMAL` or `UNKNOWN`.

DropReason

```
enum DropReason {
   OWNER_REQUEST,
   NON_OWNER_REQUEST,
   SHARING_DISABLED,
   OWNER_GONE,
   SOURCE_FCM_GONE,
   SINK_FCM_GONE,
   SOURCE_POWER_OFF,
   SINK_POWER_OFF,
   DEVICE_CONNECTION_DROPPED,
   SOURCE_DEVICE_GONE,
   SINK_DEVICE_GONE,
   RESTORE_FAILURE,
   UNKNOWN,
   IEC_BROADCAST_BROKEN
};
```

`DropReason` enumerates the situations when a Stream Manager will drop a connection and post a `ConnectionDropped` event:

- `OWNER_REQUEST` indicates that the connection has been dropped on request from the HAVi software element that created the connection.
- `NON_OWNER_REQUEST` indicates the connection was dropped on request from some other HAVi software element.
- `SHARING_DISABLED` indicates that sharing has been disabled for the source of the connection (a `PlugSharingChanged` event has been posted) and the plug owner differs from the connection owner.
- `OWNER_GONE` indicates that the owner has disappeared (detected via `MsgWatchOn`).
- `SOURCE_FCM_GONE` indicates that the source FCM has disappeared (detected via `MsgWatchOn`).
- `SINK_FCM_GONE` indicates that the sink FCM has disappeared (detected via `MsgWatchOn`).
- `SOURCE_POWER_OFF` and `SINK_POWER_OFF` indicate that the source or sink FCM, or associated DCM, has posted a `PowerStateChanged` event with `powerState` set to `False`.
- `DEVICE_CONNECTION_DROPPED` occurs if the DCM associated with a source or sink posts a `DeviceConnectionDropped` event from which it can be derived that the source and sink are no longer connected.
- `SOURCE_DEVICE_GONE` indicates that the source device has disappeared (detected after `NetworkReset`).
- `SINK_DEVICE_GONE` indicates that the sink device has disappeared (detected after `NetworkReset`).
- `RESTORE_FAILURE` occurs if the Stream Manager cannot reallocate bandwidth or the channel during connection restoration.
- `UNKNOWN` results if the Stream Manager detects that a connection is no longer present but cannot determine the cause of its removal.
- `IEC_BROADCAST_BROKEN` indicates that the connection has been dropped because IEC61883 broadcast on the connection has been broken.

ChangeReason

```
enum ChangeReason {
   STREAM_TYPE_CHANGED,
   TRANSMISSION_FORMAT_CHANGED,
   DEVICE_CONNECTION_CHANGED,
   BANDWIDTH_ADAPTATION_FAILURE,
   BANDWIDTH_ADAPTATION_SUCCESS,
};
```

`ChangeReason` enumerates the situations when a Stream Manager will post a `ConnectionChanged` event:

- `STREAM_TYPE_CHANGED` indicates that the Stream Manager has serviced a `StreamTypeChanged` event from the source FCM (and involving the source plug of the connection).
- `TRANSMISSION_FORMAT_CHANGED` indicates that the Stream Manager has serviced a `TransmissionFormatChanged` event from the source DCM (and involving the source plug of the connection).
- `DEVICE_CONNECTION_CHANGED` indicates that the Stream Manager has serviced a `DeviceConnectionChanged` event from the source or sink DCM (and involving the source or sink plug of the connection). Note, as the result of `DeviceConnectionChanged`, the stream type and/or transmission format used by the connection may also change, however only one `ConnectionChanged` event is posted.
- `BANDWIDTH_ADAPTATION_FAILURE` indicates that the Stream Manager has received a `BandwidthRequirementChanged` event from the source DCM with `allocated` set to `False`.
- `BANDWIDTH_ADAPTATION_SUCCESS` indicates that the Stream Manager has serviced a `BandwidthRequirementChanged` event from the source DCM with `allocated` set to `True`.

TransportType

```
typedef ushort TransportType;
```

Transport types are specified in Annex 11.14.

Plug

```
struct CablePlug {
   Direction  dir;
   short      cableNum;
};

struct InternalPlug {
   Direction  dir;
   ushort     fcmIndex;
   ushort     plugNum;
};

struct IecPlug {
   Direction  dir;
   short      pcrNum;
};
```

A `CablePlug` represents a non-1394 device plug (for example, an analog video plug), an `InternalPlug` represents a functional component plug, and an `IecPlug` represents an IEC

61883 device plug (a “plug control register”). A DCM plug is a `CablePlug` or `IecPlug`, an FCM plug is an `InternalPlug`.

The `InternalPlug.fcmIndex` can be inferred from the HUID of the FCM to which the plug belongs via `Fcm::GetHuid`.

The numbering for plugs starts at zero. The total number of plugs can be obtained via `Dcm::GetPlugCount` and `Fcm::GetPlugCount`.

```
union Plug switch(TransportType) {
   case CABLE:      CablePlug        cablePlug;
   case INTERNAL:   InternalPlug     internalPlug;
   case IEC61883:   IecPlug          iecPlug;
};
```

ANY_PLUG

```
typedef short PlugNumber;

const PlugNumber   ANY_PLUG   = -1;
```

`ANY_PLUG` is a special value for `IecPlug.pcrNum` and allows a client to indicate that any plug can be used for a connection.

DeviceConnection

```
struct DeviceConnection {
   Plug    src;
   Plug    dest;
};
```

TransmissionFormat

```
typedef ushort CableFormat;

struct IecFormat {
   octet[4]      FMT_FDF;
   octet[4]      mask;
};
```

`IecFormat.FMT_FDF` values correspond to those appearing in the second quadlet of the CIP header defined by IEC 61883.1, regardless of whether the CIP header is with or without the SYT field. The `mask` value indicates which bits of `FMT_FDF` should be examined when checking two `IecFormat` values for compatibility. Reserved bits, as identified by IEC 61883, shall be set to one in the mask.

```
union TransmissionFormat switch(TransportType) {
   case CABLE:      CableFormat   cableFormat;
   case INTERNAL:      ;
   case IEC61883:   IecFormat     iecFormat;
};
```

The `TransmissionFormat` is transport type dependent. It is only used for the `CABLE` and `IEC61883` transport types.

The Stream Manager APIs allow applications to specify the transmission format to use for connections or to defer selection of transmission format to the Stream Manager. When the

application specifies the transmission format, it may base its selection on formats used by existing connections (obtained via `Dcm::GetTransmissionFormat`) or those supported by DCMs (obtained via `Dcm::GetSupportedTransmissionFormats`).

PlugStatus

```
struct CablePlugStatus {
      boolean hasAttachment;
      boolean hasLegacyFcm;
};

struct InternalPlugStatus {
      boolean hasAttachment;
      boolean hasInternalConnection;
      boolean canShare;
      SEID    owner;
};

struct IecPlugStatus {
      boolean dynamicBandwidthAllocation;
      boolean hasAttachment;
};

union PlugStatus switch(TransportType) {
   case CABLE:     CablePlugStatus    cablePlugStatus;
   case INTERNAL:  InternalPlugStatus internalPlugStatus;
   case IEC61883:  IecPlugStatus      iecPlugStatus;
};
```

`PlugStatus` describes operational characteristics of a plug. The various attributes have the interpretations given below.

Cable Plugs:

- `CablePlugStatus.hasAttachment` – `True` if the plug is attached to an FCM plug.
- `CablePlugStatus.hasLegacyFcm` – `True` if the plug is used by an FCM representing an external legacy device. The Stream Manager will consider such a plug as in use.

Internal Plugs:

- `InternalPlugStatus.hasAttachment` – `True` if the plug is attached to a DCM plug.
- `InternalPlugStatus.hasInternalConnection` – `True` if the plug is connected to another FCM plug.
- `InternalPlugStatus.canShare` – `True` unless sharing is currently disabled (as a result of `Fcm::SetPlugSharing`)
- `InternalPlugStatus.owner` – if `canShare` is `False` then `owner` is the SEID of the client that disabled sharing via `Fcm::SetPlugSharing`, if `canShare` is `True` the value of `owner` is undefined and should be ignored.

IEC 61883 Plugs:

- `IecPlugStatus.dynamicBandwidthAllocation` – `True` unless static bandwidth allocation has been set via `Dcm::SetIecBandwidthAllocation`
- `IecPlugStatus.hasAttachment` – `True` if the plug is attached to an FCM plug.

Channel

```
struct CableChannel {
   CablePlug     source;
   CablePlug     sink;
};

struct IecChannel {
   ushort        isocChannel;
   IecPlug       source;
   IecPlug       sink;
};

union Channel switch(TransportType) {
   case CABLE:      CableChannel  cableChannel;
   case INTERNAL:         ;
   case IEC61883:   IecChannel    iecChannel;
};
```

IsocChannel

```
typedef ushort IsocChannel;

const IsocChannel  LEGACY_ISOC_CHANNEL   = 63;
const IsocChannel  ASYNC_STREAM_ISOC_CHANNEL   = 31;
```

FcmPlug

```
struct FcmPlug {
   TargetId      targetFcm;
   Direction     plugDir;
   ushort        plugNum;
};
```

An `FcmPlug`, like an `InternalPlug`, represents a functional component plug. While an `InternalPlug` value is only unique within the context of a DCM, an `FcmPlug` includes a `TargetId` and so is globally unique.

ConnectionId

```
struct ConnectionId {
   SEID       mgr;
   ushort     seq;
};
```

Stream Managers and applications refer to connections via globally unique connection identifiers. These values are persistent (unchanged by network resets). The `mgr` field is the SEID of the Stream Manager which created the connection, `seq` is a sequence number assigned by that Stream Manager.

ConnectionType

```
enum ConnectionType {FLOW, SPRAY, TAP};
```

Connection

```
struct Connection {
   ConnectionId       connId;
   ConnectionType     connType;
   ConnectionState    connState;
   StreamType         streamType;
   TransmissionFormat transmissionFormat;
   SEID               owner;
   FcmPlug            source;
   FcmPlug            sink;
   Channel            channel;
   uint               allocBandwidth;
};
```

The value of the transport type for a connection can be determined from the discriminator in the `Channel` or the `TransmissionFormat` value (and the discriminators must agree).

If the value of `connType` is `TAP` then the values of `source` and source plug in `channel` should be ignored. If the value of `connType` is `SPRAY` then the values of `sink` and the sink plug in `channel` should be ignored.

The value of `allocBandwidth` is in units of bits per second (bps). This value for `TAP` connections may be zero (even if a source is present via another connection). For `SPRAY` and `FLOW` connections, the value is the same for overlays as for the original connection. For `IEC61883` connections, allocated bandwidth is related to the number of allocated bandwidth units as described in section 5.9.5.4.1. If the `TransportType` of the channel is CABLE then the value of `streamType` shall be CABLE_STREAM and `TransmissionFormat` shall be CABLE_FORMAT.

StreamType

```
struct StreamType {
   StreamTypeId    id;
   boolean         constantRate;
   uint            maxBandwidth;
};
```

The value of `maxBandwidth` is in units of bits per second (bps).

`StreamType` values are returned by `Dcm::GetStreamType` and `Fcm::GetSupportedStreamTypes`. For source plugs, `maxBandwidth` indicates the maximum data rate the plug is capable of producing for the specified `StreamTypeId`. For sink plugs `maxBandwidth` indicates the maximum data rate the plug is capable of consuming for the specified `StreamTypeId`.

The `constantRate` parameter is only significant for source plugs, it indicates whether the source's bandwidth requirements are constant (e.g., DV, CD audio) or may change (e.g., a MPEG transport stream source containing several services with different bit rates).

Stream

```
struct Stream {
   boolean                 hasSource;
   FcmPlug                 source;
   sequence<ConnectionId>  connections;
```

```
};
```

A `Stream` value identifies the HAVi connections sharing a common source FCM plug, or that would share a common source FCM plug if one were connected (either directly via `FlowTo` or indirectly via `SprayOut`) to any sink FCM plug used by a member of the stream. For example, `TAP` connections using the same isochronous channel belong to the same stream.

`hasSource` is `False` when no source is present (only `TAP` connections belong to the stream). If `hasSource` is `False` the values of `source` and `streamType` are undefined and should be ignored.

ConnectionHint

```
struct ConnectionHint {
   boolean             anyTransport;
   boolean             anyStreamType;
   boolean             anyTransmissionFormat;
   boolean             anyChannel;
   TransportType       ttype;
   StreamType          stype;
   TransmissionFormat  tformat;
   Channel             channel;
};
```

`ConnectionHint` allows Stream Manager clients to specify values of various parameters needed for connection creation or to indicate that the Stream Manager is free to select their value. There are some constraints on valid `ConnectionHint` values arising from the implicit `TransportType` values in `tformat` and `channel`:

- if `anyTransport` is `True` then `anyTransmissionFormat` and `anyChannel` must also be `True`
- if `anyTransport` is `False` and `anyTransmissionFormat` is `False` then `ttype` must be the same as the transport type appearing as the discriminator in `tformat.`
- if `anyTransport` is `False` and `anyChannel` is `False` then `ttype` must be the same as the transport type appearing as the discriminator in `channel.`
- if `anyTransport` is `False` and ttype is `CABLE`, then `anyStreamType` and `anyTransmissionFormat` must be `False` and `stype` and `tformat` must be `CABLE_STREAM` and `CABLE_FORMAT` respectively.

5.9.3 Stream Manager API

StreamManager::FlowTo

Prototype

```
Status StreamManager::FlowTo(
      in boolean dynamicBw,
      in FcmPlug source,
      in FcmPlug sink,
      in ConnectionHint hint,
      out ConnectionId connId)
```

Parameters

- dynamicBw – indicates whether dynamic (dynamicBw is True) or static (dynamicBw is False) bandwidth allocation should be set
- source – a FcmPlug structure identifying a source plug
- sink – a FcmPlug structure identifying a sink plug
- hint – a ConnectionHint structure identifying possible transport type, transmission format, stream type and channel values
- connId – a ConnectionId value returned by FlowTo

Description

StreamManager::FlowTo creates a point-to-point connection between two FCM plugs. If hint.anyTransport is True the plugs are first checked for compatibility and the ability to support a connection. This will result in the determination of a transport type. CABLE transport shall not be selected in this process. And if the source and sink FCMs reside on the same device, then INTERNAL transport type shall be selected (So the selection of transport type mentioned above is actually not performed in this version of HAVi specification. But might be performed in the future version). If hint.anyTransport is False then hint.ttype is used as the transport type. If connection is possible then, depending on the transport type, one of the following will occur:

CABLE:	source is attached to a cable output plug and sink is attached to a cable input plug
INTERNAL:	an internal connection is established between source and sink via Dcm::Connect.
IEC61883:	source is attached to an oPCR and sink to an iPCR for which an IEC 61883 point-to-point connection is established.

The Stream Manager will match the lists of stream types supported by source and sink FCM plugs. This will result in a list of pairs of stream types (a, b), where a matches b and a is available at the source plug and b is available at the sink plug. (Here “available” means a supported stream type, if the plug is unconnected, or the current stream type if it is connected.) If hint.anyStreamType is True then this list is used. If hint.anyStreamType is False then the list is reduced to those pairs that match hint.stype.

The Stream Manager shall attempt to create the connection using successive stream types from the list until a connection can be successfully created or all candidate stream types have been tried. If hint.anyStreamType is False, then stream types from the final list shall be chosen so that stream types that match “exactly” with hint.stype are tried before “non-exact” matches. Two stream types match “exactly” when they are equal, so stream type A matches exactly with stream type A, stream type UNKNOWN_STREAM matches exactly with stream type UNKNOWN_STREAM, but A only matches “non-exactly” with UNKNOWN_STREAM. (Of course stream type A does not match at all with a different stream type B.)

If hint.anyTransmissionFormat is True then the Stream Manager attempts to select a transmission format for the (selected or specified) stream type that is supported by both the source and sink device. If hint.anyTransmissionFormat is False then the Stream Manager uses hint.tformat as the transmission format.

The Stream Manager will set the transmission format of the source and sink (via Dcm::SetTransmissionFormat). If the sink can autoconfigure (infer the transmission format from the incoming stream), the actual value used may be different from that set by the Stream Manager but can be read by the Stream Manager via Dcm::GetTransmissionFormat.

If hint.anyChannel is True then the Stream Manager selects an unused channel, otherwise it uses the specified channel. If hint.anyChannel is True when transport type is CABLE, the EANY_CHANNEL error is returned. For IEC61883 connections, if channel refers to an existing

point-to-point connection, an overlay is attempted; otherwise the specified `channel` is allocated.

For `IEC61883` connections, if a DCM plug number within `channel` is `ANY_PLUG` then the Stream Manager is free to select which device plug to use, else it attempts to use the specified plug.

For `IEC61883` connections, if the stream type assigned to the source is variable rate (`constantRate` is `False`) then `Dcm::SetIecBandwidthAllocation` is called as follows:

- If `dynamicBw` is `False` and `hint.anyStreamType` is `True`, then the Stream Manager shall call `Dcm::SetIecBandwidthAllocation` using the `maxBandwidth` value associated with the stream type of the source plug.

- If `dynamicBw` is `False` and `hint.anyStreamType` is `False`, then the Stream Manager shall call `Dcm::SetIecBandwidthAllocation` using `hint.stype.maxBandwidth`.

- In any other case (i.e., `dynamicBw` is `True`), then the Stream Manager shall call `Dcm::SetIecBandwidthAllocation` with `maxBandwidth` set to 0 (dynamic bandwidth allocation). If this call fails (because a previous connection has set static bandwidth allocation) then the Stream Manager shall continue in connection establishment using static bandwidth allocation.

Note that utilization of a `CABLE` connection would typically require that a physical cable join the DCM plugs. The Stream Manager is not responsible for maintaining information about physical cabling configurations. When it receives a `FlowTo` request for a `CABLE` connection it simply attaches the FCM plugs to the DCM plugs at each end of the connection.

Error codes

- `StreamManager::ESOURCE_FCM` – the FCM indicated by `source` does not exist
- `StreamManager::ESINK_FCM` – the FCM indicated by `sink` does not exist
- `StreamManager::ESOURCE_PLUG` – the FCM indicated by `source` does not contain the specified plug
- `StreamManager::ESINK_PLUG` – the FCM indicated by `sink` does not contain the specified plug
- `StreamManager::EUNSUP_TRANSPORT` – connection requires an unsupported transport type
- `StreamManager::EUNSUP_STREAM` – connection requires an unsupported stream type
- `StreamManager::ENO_MATCH_STREAM` – the plugs are incompatible (stream type mismatch)
- `StreamManager::ENO_MATCH_FMT` – the plugs are incompatible (transmission format mismatch)
- `StreamManager::ENO_MATCH_BW` – the source bandwidth exceeds that supported by the sink or `hint.Stype.maxBW` exceeds supported `Stype.maxBW` of source/sink FCM (bandwidth mismatch)
- `StreamManager::ENO_MATCH_SPEED` – the source uses a transmission speed unsupported by the sink
- `StreamManager::ENO_MATCH_TRANSPORT` – the plugs are incompatible (transport type mismatch)
- `StreamManager::ENO_MATCH_DIR` – the plugs are incompatible (direction mismatch)
- `StreamManager::EINVALID_FMT` – the specified transmission format is not supported by the source or sink
- `StreamManager::ESOURCE_BUSY` – the source plug is a member of another stream
- `StreamManager::ESINK_BUSY` – the sink plug is a member of another stream
- `StreamManager::EDEV_BUSY` – failure to allocate a device plug
- `StreamManager::EINSUFF_BANDWIDTH` – bandwidth allocation has failed

- StreamManager::EINSUFF_CHANNEL – channel allocation has failed
- StreamManager::EINVALID_CHANNEL – the specified channel value is not valid
- StreamManager::ECHANNEL_BUSY – hint.anyChannel is False and the specified channel is in use by a different source
- StreamManager::EASYNC_CHANNEL – channel is set to ASYNC_STREAM_ISOC_CHANNEL
- StreamManager::ESTATICBW – dynamicBw has the value False, the stream type is variable rate, but the source cannot be set to static bandwidth allocation
- StreamManager::ERESERVED_SINK – the FCM indicated by sink is reserved (and not by the software element making the FlowTo request)
- StreamManager::EDEV_CONN – failure to establish a device connection
- StreamManager::ESHARE – the connection cannot be established because the source plug is not sharable (and the owner differs from the software element making the FlowTo request)
- StreamManager::EANY_CHANNEL– hint.anyChannel is True when transportType is CABLE.
- StreamManager::ERESERVED_SOURCE– required connection need to be established (i.e., not overlay) and rejected due to reservation protection.

StreamManager::SprayOut

Prototype

```
Status StreamManager::SprayOut(
      in boolean dynamicBw,
      in FcmPlug source,
      in ConnectionHint hint,
      out ConnectionId connId)
```

Parameters

- dynamicBw – indicates whether dynamic (dynamicBw is True) or static (dynamicBw is False) bandwidth allocation should be set
- source – a Plug structure identifying a source plug
- hint – a ConnectionHint structure identifying possible transport type, transmission format, stream type and channel values
- connId – a connection identifier

Description

StreamManager::SprayOut creates an unbound connection for a source plug (a connection of type SPRAY). Depending on the transport type, one of the following will occur:

CABLE: source is attached to a cable output plug.

INTERNAL: the EINVALID_PARAMETER error is returned.

IEC61883: source is attached to an oPCR for which an IEC 61883 broadcast-out connection is established. Note that IEC 61883 restrictions on establishment of broadcast connections may prevent the Stream Manager from creating a SPRAY connection despite the availability of resources. If so the Stream Manager shall attempt to establish the broadcast-out connection via Dcm::IecSprayOut. If this API is not supported by the DCM then the StreamManager::EIEC61883 error is returned. If Dcm::IecSprayOut returns Dcm::EINSUFF_BANDWIDTH or Dcm::EINSUFF_CHANNEL then the Stream Manager returns StreamManager::EINSUFF_BANDWIDTH or StreamManager::EINSUFF_CHANNEL respectively. If Dcm::IecSprayOut succeeds the Stream Manager will set the connection's Connection.allocBandwidth to the value from the oPCR.

If hint.anyTransport is True the EINVALID_PARAMETER error is returned.

The Stream Manager will construct a list of stream types available at the source FCM plug. (Here "available" means a supported stream type, if the plug is unconnected, or the current stream type if it is connected.) If hint.anyStreamType is True then this list is used. If hint.anyStreamType is False then the list is reduced to those entries that match hint.stype.

The Stream Manager shall attempt to create the connection using successive stream types from the list until a connection can be successfully created or all candidate stream types have been tried. If hint.anyStreamType is False, then stream types from the final list shall be chosen so that stream types that match "exactly" with hint.stype are tried before "non-exact" matches. Two stream types match "exactly" when they are equal, so stream type A matches exactly with stream type A, stream type UNKNOWN_STREAM matches exactly with stream type UNKNOWN_STREAM, but A only matches "non-exactly" with UNKNOWN_STREAM. (Of course stream type A does not match at all with a different stream type B.)

If hint.anyTransmissionFormat is True then the Stream Manager attempts to select a transmission format for the (selected or specified) stream type that is supported by the source device. If hint.anyTransmissionFormat is False then the Stream Manager uses hint.tformat as the transmission format.

The Stream Manager will set the transmission format of the source (via Dcm::SetTransmissionFormat).

If hint.anyChannel is True then the Stream Manager selects an unused channel, otherwise it uses the specified channel. If hint.anyChannel is True when transport type is CABLE, the EANY_CHANNEL error is returned. For IEC61883 connections, if channel refers to an existing point-to-point connection, an overlay is attempted; otherwise the specified channel is allocated.

For IEC61883 connections, if a DCM plug number within channel is ANY_PLUG then the Stream Manager is free to select which DCM plug to use, else it attempts to use the specified plug. The sink field in channel should be ignored.

For IEC61883 connections, if the stream type assigned to the source is variable rate (constantRate is False) then Dcm::SetIecBandwidthAllocation is called as follows:

- If dynamicBw is False and hint.anyStreamType is True, then the Stream Manager shall call Dcm::SetIecBandwidthAllocation using the maxBandwidth value associated with the stream type of the source plug.

- If dynamicBw is False and hint.anyStreamType is False, then the Stream Manager shall call Dcm::SetIecBandwidthAllocation using hint.stype.maxBandwidth.

- In any other case (i.e., dynamicBw is True), then the Stream Manager shall call Dcm::SetIecBandwidthAllocation with maxBandwidth set to 0 (dynamic bandwidth allocation). If this call fails (because a previous connection has set static bandwidth allocation) then the Stream Manager shall continue in connection establishment using static bandwidth allocation.

Error codes

- StreamManager::ESOURCE_FCM – the FCM indicated by source does not exist
- StreamManager::ESOURCE_PLUG – the FCM indicated by source does not contain the specified plug

- StreamManager::EUNSUP_TRANSPORT – the transport type indicated by channel is not supported
- StreamManager::ENO_MATCH_DIR – the plug is not a source (output) plug
- StreamManager::EUNSUP_STREAM – connection requires an unsupported stream type
- StreamManager::ENO_MATCH_STREAM – the plug does not support the specified stream type or, if already connected, is set to another stream type
- StreamManager::EINVALID_FMT – the specified transmission format is not supported by the source
- StreamManager::ESOURCE_BUSY – the plug is a member of another stream
- StreamManager::EDEV_BUSY – failure to allocate a device plug
- StreamManager::EINSUFF_BANDWIDTH – bandwidth allocation has failed
- StreamManager::EINSUFF_CHANNEL – channel allocation has failed
- StreamManager::EINVALID_CHANNEL – the specified channel value is not valid
- StreamManager::ECHANNEL_BUSY – hint.anyChannel is False and the specified channel is in use by a different source
- StreamManager::EASYNC_CHANNEL – channel is set to ASYNC_STREAM_ISOC_CHANNEL
- StreamManager::ESTATICBW – dynamicBw has the value False, the stream type is variable rate, but the source cannot be set to static bandwidth allocation
- StreamManager::EBROADCAST – unable to establish the connection because of Dcm::IecSprayOut is not supported by the source DCM.
- StreamManager::EDEV_CONN – failure to establish a device connection
- StreamManager::ESHARE – the connection cannot be established because the source plug is not sharable (and the owner differs from the software element making the SprayOut request)
- StreamManager::EANY_CHANNEL– hint.anyChannel is True when transportType is CABLE.
- StreamManager::ERESERVED_SOURCE– required connection need to be established (i.e., not overlay) and rejected due to reservation protection.
- StreamManager::ENO_MATCH_FMT– the specified TransmissionFormat does not match with that of already existing connection.

StreamManager::TapIn

Prototype

```
Status StreamManager::TapIn(
      in FcmPlug sink,
      in ConnectionHint hint,
      out ConnectionId connId)
```

Parameters

- sink – a FcmPlug structure identifying sink plug
- hint – a ConnectionHint structure identifying possible transport type, transmission format, stream type and channel values
- connId – a connection identifier

Description

StreamManager::TapIn creates an unbound connection for an FCM sink plug (a connection of type TAP). Depending on the transport type, one of the following will occur:

CABLE:	sink is attached to a cable input plug.
INTERNAL:	the EINVALID_PARAMETER error is returned.
IEC61883:	sink is attached to an iPCR for which an IEC 61883 broadcast-in connection is established. Note that IEC 61883 restrictions on establishment

of broadcast connections may prevent the Stream Manager from creating a `TAP` connection despite the availability of resources. If so the Stream Manager shall attempt to establish the broadcast-in connection via `Dcm::IecTapIn`. If this API is not supported by the DCM then the `StreamManager::EIEC61883` error is returned.

If `hint.anyTransport` is `True` the `EINVALID_PARAMETER` error is returned.

The Stream Manager will construct a list of stream types available at the sink FCM plug. (Here "available" means a supported stream type, if the plug is unconnected, or the current stream type if it is connected.) If `hint.anyStreamType` is `True` then this list is used. If `hint.anyStreamType` is `False` then the list is reduced to those entries that match `hint.stype`.

The Stream Manager shall attempt to create the connection using successive stream types from the list until a connection can be successfully created or all candidate stream types have been tried. If `hint.anyStreamType` is `False`, then stream types from the final list shall be chosen so that stream types that match "exactly" with `hint.stype` are tried before "non-exact" matches. Two stream types match "exactly" when they are equal, so stream type A matches exactly with stream type A, stream type `UNKNOWN_STREAM` matches exactly with stream type `UNKNOWN_STREAM`, but A only matches "non-exactly" with `UNKNOWN_STREAM`. (Of course stream type A does not match at all with a different stream type B.)

If `hint.anyTransmissionFormat` is `True` then the Stream Manager attempts to select a transmission format for the (selected or specified) stream type that is supported by the sink device. If `hint.anyTransmissionFormat` is `False` then the Stream Manager uses `hint.tformat` as the transmission format.

The Stream Manager will set the transmission format of the sink (via `Dcm::SetTransmissionFormat`). If the sink can autoconfigure (infer the transmission format from the incoming stream), the actual value used may be different from that set by the Stream Manager but can be read by the Stream Manager via `Dcm::GetTransmissionFormat`.

If `hint.anyChannel` is `True` then the Stream Manager selects an unused channel, otherwise it uses the specified channel. If `hint.anyChannel` is `True` when transport type is `CABLE`, the `EANY_CHANNEL` error is returned.

For `IEC61883` connections, if a plug number within `channel` is `ANY_PLUG` then the Stream Manager is free to select which DCM plug to use, else it attempts to use the specified plug. The `source` field in `channel` should be ignored.

The Stream Manager shall assure that the connection does not belong to a stream for which sharing is disabled (by a software element other than the `TapIn` client). If this is not possible `StreamManager::ESHARE` is returned.

Error codes

- `StreamManager::ESINK_FCM` – the FCM indicated by `sink` does not exist
- `StreamManager::ESINK_PLUG` – the FCM indicated by `sink` does not contain the specified plug
- `StreamManager::EUNSUP_TRANSPORT` – the transport type indicated by `channel` is not supported
- `StreamManager::ENO_MATCH_DIR` – the plug is not a sink (input) plug
- `StreamManager::EUNSUP_STREAM` – connection requires an unsupported stream type
- `StreamManager::ENO_MATCH_STREAM` – the plug does not support the specified stream type or, if already connected, is set to another stream type

- `StreamManager::EINVALID_FMT` – the specified transmission format is not supported by the sink
- `StreamManager::ESINK_BUSY` – the plug is a member of another stream
- `StreamManager::EDEV_BUSY` – failure to allocate a device plug
- `StreamManager::EINSUFF_CHANNEL` – `hint.anyChannel` is `True` but all channels are in use
- `StreamManager::EASYNC_CHANNEL` – `channel` is set to `ASYNC_STREAM_ISOC_CHANNEL`
- `StreamManager::EINVALID_CHANNEL` – the specified `channel` value is not valid
- `StreamManager::EBROADCAST` – unable to establish the connection because `Dcm::IecTapIn` is not supported by the sink DCM.
- `StreamManager::ERESERVED_SINK` – the FCM indicated by `sink` is reserved (and not by the software element making the `TapIn` request)
- `StreamManager::EDEV_CONN` – failure to establish a device connection
- `StreamManager::ESHARE` – the connection cannot be established because if the connection were established it would be a member of a stream for which the source plug is not sharable (and the owner differs from the software element making the `TapIn` request)
- `StreamManager::EANY_CHANNEL`– `hint.anyChannel` is `True` when `transportType` is `CABLE`.
- `StreamManager::ENO_MATCH_FMT`– the specified `TransmissionFormat` does not match with that of already existing connection.

StreamManager::Drop

Prototype

```
Status StreamManager::Drop(in ConnectionId connId)
```

Parameters

- `connId` – a valid connection identifier

Description

`StreamManager::Drop` breaks a connection. Resources associated with the connection are released provided no plugs are bound to the underlying channel. The SEID of the issuer of `StreamManager::Drop` need not be the same as that of the owner of the connection (i.e., it is possible to preempt connections). However only trusted or system SEIDs can drop connections which they do not own. When sink FCM is reserved then `ERESERVED_SINK` is returned and connection will be kept.

Error codes

- `StreamManager::ECONN_ID` – `connId` does not refer to a connection created by the Stream Manager
- `StreamManager::EACCESS_VIOLATION` – an untrusted software element has attempted to drop a connection which it does not own
- `StreamManager::ERESERVED_SINK` – the FCM indicated by sink is reserved (and not by the software element making the Drop request)

StreamManager::GetLocalConnectionMap

Prototype

```
Status StreamManager::GetLocalConnectionMap(
      out sequence<Connection> list)
```

Parameters

- `list` – a list of connections managed by the Stream Manager. The safe parameter size limit is 32 `Connection` values.

Description

`StreamManager::GetLocalConnectionMap` queries the Stream Manager for all connections which it manages (i.e., has created).

StreamManager::GetGlobalConnectionMap

Prototype

```
Status StreamManager::GetGlobalConnectionMap(
      out sequence<Connection> list)
```

Parameters

- `list` – a list of all HAVi connections on the home network. The safe parameter size limit is 128 `Connection` values.

Description

`StreamManager::GetGlobalConnectionMap` queries the Stream Manager for a list of all HAVi connections on the home network. The Stream Manager may implement this function by combining the results of `GetLocalConnectionMap` queries to all other Stream Managers on the network, with the connections it controls itself. There is no guarantee that connections are not created or dropped while the Stream Manager is processing the query.

Error codes

- `StreamManager::ENETWORK` – the response may not be complete due to a network problem

StreamManager::GetConnection

Prototype

```
Status StreamManager::GetConnection(
      in ConnectionId connId,
      out Connection conn)
```

Parameters

- `connId` – a valid connection identifier
- `conn` – the `Connection` structure for `connId`

Description

`StreamManager::GetConnection` queries the Stream Manager for the `Connection` data structure associated with `connId`.

Error codes

- `StreamManager::ECONN_ID` – `connId` does not refer to a connection created by the Stream Manager

StreamManager::GetStream

Prototype

```
Status StreamManager::GetStream(
      in ConnectionId connId,
      out Stream stream)
```

Parameters

- `connId` – a valid connection identifier
- `stream` – the `Stream` structure for `connId`. The safe parameter size limit for `stream.connections` is 32 `Connection` values.

Description

`StreamManager::GetStream` requests the Stream Manager to build the `Stream` data structure that contains the connection specified by `connId`. Building the `Stream` structure may involve communication (via `GetLocalConnectionMap`) with other Stream Managers.

Error codes

- `StreamManager::ECONN_ID` – `connId` does not refer to a connection created by any Stream Manager

`StreamManager::ENETWORK` – the response may not be complete due to a network problem

5.9.4 Stream Manager Events

ConnectionAdded

```
void ConnectionAdded(in ConnectionId connId)
```

Parameters

- `connId` – the connection identifier of the new connection

Description

The `ConnectionAdded` event is issued by a Stream Manager after a successfully servicing a `FlowTo`, `SprayOut` or `TapIn` request.

ConnectionDropped

```
void ConnectionDropped(
      in ConnectionId connId,
      in sequence<DropReason> reasons)
```

Parameters

- `connId` – the connection identifier of the dropped connection
- `reason` – the reasons the connection was dropped, the safe parameter size limit is 4 `DropReason` values.

Description

The `ConnectionDropped` event is issued by a Stream Manager after it has dropped a connection.

ConnectionChanged

```
void ConnectionChanged(
      in ConnectionId connId,
      in ConnectionState oldState,
      in ConnectionState newState,
      in ChangeReason reason)
```

Parameters

- `connId` – the connection identifier of the changed connection
- `oldState` – the state of the connection before the change
- `newState` – the state of the connection after the change
- `reason` – the reason the connection has changed

Description

The `ConnectionChanged` event is issued by a Stream Manager after it has handled a `StreamTypeChanged`, `TransmissionFormatChanged`, `BandwidthRequirementChanged`, or `DeviceConnectionChanged` event involving the connection. The change may have been handled successfully or not, as indicated by the `newState` parameter.

5.9.5 Stream Manager Procedures

5.9.5.1 Stream Type Matching

The following pseudo-code describes the procedure for determining when the stream types associated with a source plug and a sink plug "match" (or whether a requested stream type matches one already associated with a plug).

Let `S1` and `S2` be two `StreamTypeId` values that are to be tested for a match. Then there are two main cases:

Case I: both `S1` and `S2` are HAVi stream types

```
if(S1.typeNo == S2.typeNo)
   return ok
else if(S1.typeNo == UNKNOWN_STREAM ||
        S2.typeNo == UNKNOWN_STREAM)
   return ok
else if((S1.typeNo.major == S2.typeNo.major) &&
        (S1.typeNo.minor == UNKNOWN_STREAM ||
         S2.typeNo.minor == UNKNOWN_STREAM))
   return ok
else
   return ENO_MATCH_STREAM
```

Case II: either `S1` or `S2` (or both) are non-HAVi (i.e. vendor) stream types

```
if(S1 == S2)
   return ok
else if (S1.vendorId == 0 && S1.typeNo == UNKNOWN_STREAM)
   return ok
else if (S2.vendorId == 0 && S2.typeNo == UNKNOWN_STREAM)
   return ok
else
   return ENO_MATCH_STREAM
```

5.9.5.2 Transmission Format Matching

The following pseudo-code describes the procedure for determining when two transmission formats "match". Let TF1 and TF2 be two TransmissionFormat values that are to be tested for a match.

```
if TF1 and TF2 do not use the same transport type
      return ENO_MATCH_FMT

switch on the transport type {
case CABLE:
      if TF1 == TF2
         return ok
      else
         return ENO_MATCH_FMT
case INTERNAL:
         return ok
case IEC61883:
      if(TF1.FMT_FDF & (TF1.mask & TF2.mask) ==
         TF2.FMT_FDF & (TF1.mask & TF2.mask))
            return ok
      else
            return ENO_MATCH_FMT
}
```

5.9.5.3 NO_SIGNAL Stream Type

NO_SIGNAL stream type represents that no valid signal is generated by the source FCM. (e.g., empty packets going on IEC connection). For the purpose of NO_SIGNAL is just to inform the state, Dcm::SetStreamTypeId with specifying NO_SIGNAL always results Dcm::ENOT_SUPPORTED. And NO_SIGNAL will not be returned by Dcm::GetAvailableStreamTypes. It can be returned by Fcm::GetSupportedStreamTypes or Dcm::GetStreamType API from source DCM or carried by events generated by source plug. Thus the Stream Manager which involved by a StreamTypeChanged with NO_SIGNAL will always fail to match the stream types, so the stream manager will change ConnectionState to FAILURE and FailureReason to STREAM_TYPE_MATCH_FAILURE then generate ConnectionChanged event.

Note: It is not mandatory to support NO_SIGNAL stream type even if an FCM is capable of producing a signal which is not valid. Also, it depends on the implementation of the FCM as to which stream is valid. So stream types other than NO_SIGNAL do not guarantee that a connection is carrying valid content for user presentation.

5.9.5.4 IEC 61883 Connections

5.9.5.4.1 Bandwidth Allocation

For IEC 61883 connections, the bandwidth computation procedure used by the Stream Manager is:

1) Determine the *payload* – the maximum number of quadlets in one isochronous packet excluding headers. If the DCM has dynamic bandwidth allocation set (dynamicBw is True in FlowTo or SprayOut) then obtain the *payload* from the oPCR. If the DCM has static bandwidth allocation set (dynamicBw is False) either obtain the payload from the oPCR or calculate via *payload* = maxBandwidth / (*8000* * *32*). (Note – when an oPCR uses static bandwidth

allocation these two values are the same.)

2) Select the *overhead_ID* to be used by the oPCR. (HAVi does not specify how this value is selected.)
3) Determine the maximum packet speed between the source device and the sink device (for example, by consulting the IEEE 1394 speed map). From this determine the *data rate coefficient (DR)*. In case there is no sink, e.g., a `SPRAY` connection, the data rate capability of the source shall be used.
4) Knowing *payload*, *overhead_ID* and *DR* calculate the number of bandwidth allocation units, *BWU*, using the formula in IEC 61883.1.
5) In case of IEC broadcast-out connections established via `DCM::IecSprayOut`, the DCM/device shall allocate *BWU* from the 1394 Isochronous Resource Manager. In all other cases, the Stream Manager shall allocate *BWU*.

5.9.5.4.2 Static and Dynamic Bandwidth Allocation

Certain devices are inherently variable data rate – their bandwidth requirements may change depending upon content. An example is a digital tuner changing from a low bandwidth service to a high bandwidth service with the same `StreamTypeId`. To accommodate such devices and encourage efficient use of the 1394 network, HAVi provides two bandwidth allocation strategies: *static bandwidth allocation* and *dynamic bandwidth allocation*. Stream Manager clients indicate their bandwidth allocation preference when requesting a connection to be established. Overlays will not change the method used.

- static bandwidth allocation – Stream Managers allocate bandwidth corresponding to the payload in the oPCR. If the source is variable rate the Stream Manager first requests the DCM to set static bandwidth allocation (via `Dcm::SetIecBandwidthAllocation`), if this request fails then the Stream Manager returns the `ESTATICBW` error. If bandwidth requirements change it is the responsibility of the DCM to issue the `BandwidthRequirementChanged` event.

- dynamic bandwidth allocation – Stream Managers allocate bandwidth corresponding to the payload in the oPCR. If the source is variable rate the Stream Manager first requests the DCM to set dynamic bandwidth allocation via `Dcm::SetIecBandwidthAllocation`. (If this request fails then static bandwidth allocation is used.) If bandwidth requirements change it is the responsibility of the DCM to assure that bandwidth allocation is attempted, or excessive bandwidth is released, (whether by the device or itself) and issue the `BandwidthRequirementChanged` event.

Note that even in the case of static bandwidth allocation, the bandwidth guarantee may not be absolute. If the `maxBandwidth` the Stream Manager specifies in `Dcm::SetIecBandwidthAllocation` is the maximum value for just the current stream type (as opposed to all possible stream types of which the source is capable) then as a result of a `StreamTypeChanged` or `DeviceConnectionChanged` event, the value of `StreamType.maxBandwidth` associated with source FCM plug may increase and no longer correspond to the allocated bandwidth as indicated in the source oPCR payload (in which case the connection will enter the state `BANDWIDTH_FAILURE`).

5.9.5.4.3 Overlays

`FlowTo`, `TapIn` or `SprayOut` may result in an IEC 61883 overlay connection. Conditions for performing an overlay when handling a `FlowTo` are:

- the source FCM plug is attached to an connected oPCR (where “connected” is as defined by IEC 61883.1). If not then an overlay is not necessary and establishment of a point-to-point connection should be attempted.

- the sink FCM plug is unattached, or attached to an unconnected iPCR, or attached to an iPCR connected to the oPCR (where “connected” and “unconnected” are as defined by IEC 61883.1). If not then `ESINK_BUSY` should be returned.

- the maximum packet speed between the source and sink devices is greater than or equal to the data rate specified in the oPCR. If not then `ENO_MATCH_SPEED` is returned.

- the stream type ID requested by the client and associated with the source and sink can be matched. If not then `ENO_MATCH_STREAM` should be returned.

- the transmission format requested by the client and associated with the source and sink can be matched. If not then `ENO_MATCH_FMT` should be returned.

- the maximum bandwidth of the source should not exceed the maximum bandwidth of the sink. If not then `ENO_MATCH_BW` should be returned.

Conditions for performing an overlay when handling a `SprayOut` are:

- `hint.anyChannel` is `True` or `hint.channel` contains an `IecChannel` value, and this value refers to a currently allocated 1394 channel.

- the source FCM plug is attached to an active oPCR (where “active” is as defined by IEC 61883.1). If not then an overlay is not necessary and a establishment of a broadcast-out connection should be attempted.

- the stream type ID requested by the application matches that in use by the source FCM plug. If not then `ENO_MATCH_STREAM` should be returned.

- the transmission format requested by the application matches that is use by the oPCR. If not then `ENO_MATCH_FMT` should be returned.

Conditions for performing an overlay when handling a `TapIn` are:

- `hint.anyChannel` is `True` or `hint.channel` contains an `IecChannel` value, and this value refers to a currently allocated 1394 channel.

- the sink FCM plug is either unattached, or attached to an unconnected iPCR, or attached a connected iPCR using the channel referred to by the `IecChannel` value (where “connected” and “unconnected” are as defined by IEC 61883.1). If not then `ESINK_BUSY` should be returned.

- the stream type ID requested by the application is supported by the sink FCM plug (if unattached) of matches that used by the sink FCM plug (if attached). If not then `ENO_MATCH_STREAM` should be returned.

- the transmission format requested by the application matches that is use by the iPCR. If not then `ENO_MATCH_FMT` should be returned.

5.9.5.4.4 Usage of Special Channels

Many legacy 1394 devices by default allocate isochronous channel 63 (`LEGACY_ISOC_CHANNEL`). In addition channel 31 (`ASYNC_STREAM_ISOC_CHANNEL`) is used by IEEE 1394a devices for asynchronous streaming. To avoid possible conflict with the operation of these devices a HAVi Stream Manager shall restrict its use of channel 31 and 63 as follows:

- a Stream Manager shall not use channel 31 for a HAVi connection
- a Stream Manager may overlay on channel 63, however a Stream Manager shall only allocate channel 63 for a HAVi connection if the `Channel.isocChannel` parameter of `FlowTo` or `SprayOut` contains `LEGACY_ISOC_CHANNEL` and `hint.anyChannel` is `False`, i.e., a Stream Manager shall not allocate channel 63 if the `hint.anyChannel` parameter of `FlowTo` or `SprayOut` is `True`

5.10 Resource Manager

5.10.1 Services Provided

Service	Comm Type	Locality	Access
ResourceManager::Reserve	M	global	all
ResourceManager::Release	M	global	all
ResourceManager::Negotiate	M	global	all
<Client>::PreemptionRequest	MB	global	Resource Managers
ResourceManager::ScheduleAction	M	global	all
ResourceManager::UnscheduleAction	M	global	all
ResourceManager::GetLocalScheduledActions	M	global	all
ResourceManager::GetScheduledActionData	M	global	all
ResourceManager::TriggerNotification	M	global	all
<Client>::AwakeNotification	MB	global	Resource Managers
ResourceManager::GetScheduledConnections	M	global	Resource Managers
InvalidScheduledAction	E	global	all
AbortedScheduledAction	E	global	all
ErroneousScheduledAction	E	global	all

5.10.2 Resource Manager Data Structures

ClientRole

Definition

```
enum ClientRole {USER, SYSTEM };
```

Description

The two roles of a client.

ReservationResult

```
enum ReservationResult {
      FAILED_OTHER, PRIMARY, SECONDARY,
      FAILED_SELF, NOT_SUPPORTED, NO_RESOURCE
};
```

NegotiationResult

```
enum NegotiationResult {
      ACCEPTED, REJECTED, TIMEOUT, NOT_SUPPORTED,
      NO_RESOURCE, SYSTEM_CLIENT
};
```

ResourceRequestRecord

Definition

```
struct ResourceRequestRecord {
      SEID    resource;
      boolean primary;
};
```

Description

A specification of a resource request record. `resource` specifies the resource to be reserved by the resource management system. `primary` is `True` if the resource should be reserved with primary access rights; otherwise it should be reserved with secondary access rights.

ResourceStatusRecord

Definition

```
struct ResourceStatusRecord {
      SEID                resource;
      ReservationResult   acquisition;
};
```

Description

A specification of a resource status record. `resource` specifies the resource for which data is retrieved by the resource management system. `acquisition` specifies the result of a reservation attempt:

- `FAILED_OTHER` – no access rights because for at least one other specified resource no access rights could be acquired
- `PRIMARY` – primary access rights acquired
- `SECONDARY` – secondary access rights acquired
- `FAILED_SELF` – no access rights because the requested access rights could not be acquired
- `NOT_SUPPORTED` – the resource does not support reservation
- `NO_RESOURCE` – the resource does not exist (or an invalid SEID value)

ResourceNegotiateRecord

Definition

```
struct ResourceNegotiateRecord {
      SEID               resource;
      NegotiationResult  result;
      wstring<50>        info;
};
```

Description

A specification of a resource negotiate record. `resource` specifies the resource for which negotiation was performed by the resource management system. `result` specifies the result of a preemption request to the primary client that holds the resource (if any):

- `ACCEPTED` – accepted (also if there is no primary client)
- `REJECTED` – rejected by the client
- `TIMEOUT` – timeout has expired
- `NOT_SUPPORTED` – the client does not support preemption requests
- `NO_RESOURCE` – the resource does not exist (or an invalid SEID value), or does not support reservation
- `SYSTEM_CLIENT` – negotiation with a system client is not allowed

`info` specifies an information string that may be supplied by the current client of the resource.

SAReference

Definition

```
struct SAReference {
      SEID  actionScheduler;
      long  index;
};
```

Description

References a Scheduled Action within the network.

Command

Definition

```
struct Command {
      OperationCode    opCode;
      sequence<octet>  command;
      HUID             huid;
};
```

Parameters

- `opCode` – the operation code of the command sent to the FCM or DCM
- `command` – sequence of octets that will directly be addressed to the FCM or DCM; this field will be the bytes after the `TransactionId` field in the message representation discussed in section 3.2.3.2 (Figure 14)
- `huid` – HUID of the FCM or DCM that will execute the command

Description

Defines a single command associated with the HUID of the FCM or DCM that will have to perform it. The command is a sequence of octets the Action Scheduler does not have to interpret.

SAConnection

Definition

```
struct SAConnection {
      FcmPlug       source;
      FcmPlug       sink;
      StreamType    stype;
};
```

Parameters

`FcmPlug` and `StreamType` types are defined within the Stream Manager.

Description

Defines a connection structure that is maintained in each Action Scheduler to perform the scheduled bandwidth reservation.

SAPeriod

Definition

```
enum SAPeriod {NONE, DAILY, WEEKLY};
```

Description

Refers to the periodicity parameter defining the way a Scheduled Action must be repeated. `NONE` means that the Scheduled Action must only be performed once.

RMConnection

Definition

```
struct RMConnection {
      SAConnection  connection;
      DateTime      startTime;
      DateTime      stopTime;
      SAPeriod      periodicity;
};
```

Description

Structure to exchange connection information between Resource Managers.

5.10.3 Resource Manager API

ResourceManager::Reserve

Prototype

```
Status ResourceManager::Reserve(
      in boolean preemptive,
      in ClientRole role, in wstring<50> info,
      in OperationCode preemptionRequestCode,
      in sequence<ResourceRequestRecord> requestRecords,
      out sequence<ResourceStatusRecord> statusRecords)
```

Parameters

- `preemptive` – if `True`, the resources are preempted from the current resource clients; otherwise, a non-intrusive reservation for the resources is attempted
- `role` – the requester is a user or system client
- `info` – an information string related to this reservation
- `preemptionRequestCode` – the operation code of the client's `PreemptionRequest` method (may be invoked by Resource Managers during negotiation)
- `requestRecords` – the request records of the resource group to be reserved. The safe parameter size limit is 20 `ResourceRequestRecord` values.
- `statusRecords` – the returned status records of the resource group. The safe parameter size limit is 20 `ResourceStatusRecord` values.

Description

A resource group preemption is done, or a non-intrusive reservation is attempted on behalf of the invoking contender. The request records specify for each resource whether primary or secondary access rights are required. The returned status records specify the result for each resource in the request record list. The value of `role` should reflect the role of the requesting contender. `info` is an information string recorded in the reserved resources upon a successful reservation.

A Resource Manager should use `Fcm::GetReservationStatus` messages to learn about the status of FCMs. It then analyzes the results to determine whether the requested type of reservation (non-intrusive or preemptive) will go through or not. (For rules regarding FCM reservation, refer to the section on `Fcm::Reserve`.) If it will go through, then it sends `Reserve` commands to each of the FCMs. If it turns out that due to contention from other requests or a network failure an FCM reservation request is unable to go through, then the RM releases all FCMs in the request which have been reserved so far. In case of failure to reserve all resources in the group, the `acquisition` field in the `ResourceStatusRecord` for each resource which succeeded will be `FAILED_OTHER`.

A client is recommended to supply the same value for `PreemptionRequestCode` if it does several separate reservations. This is recommended because `ResourceManager::Negotiate` can only pick one of the supplied codes in a possible future `<Client>::PreemptionRequest` to the client.

Error codes

- `ResourceManager::ERESERVE_FAILED` – the reservation failed

ResourceManager::Release

Prototype

```
Status ResourceManager::Release(
      in sequence<SEID> resources,
      in boolean neutral)
```

Parameters

- `resources` – the resource group to be released. The safe parameter size limit is 20 `SEID` values.
- `neutral` – if `True` FCMs will return to a neutral state before they are released

Description

Any subset of the reserved resources of the invoking client will be released, provided the client has reserved them. Invalid SEID values are ignored.

ResourceManager::Negotiate

Prototype

```
Status ResourceManager::Negotiate(
      in boolean request, in ClientRole role,
      in long negotiateTimeout,
      in wstring<50> info, in sequence<SEID> resources,
      out sequence<ResourceNegotiateRecord> negotiateRecords)
```

Parameters

- `request` – if `True`, this is a preemption request; otherwise it is a withdrawal
- `role` – the requester is a user or system client
- `negotiateTimeout` – the period in seconds the negotiation may last
- `info` – an information string related to this negotiation
- `resources` – the resources for which negotiation takes place. The safe parameter size limit is 20 `SEID` values.
- `negotiateRecords` – the returned negotiate records for the resources. The safe parameter size limit is 20 `ResourceNegotiateRecord` values.

Description

A resource group negotiation is done on behalf of the invoking contender. The value of `role` should reflect the role of the requesting contender. `info` is an information string used to inform current resource clients about the negotiation. `negotiateTimeout` specifies the amount of time the negotiation may take place. The returned negotiate records specify the result for each resource.

`role`, `negotiateTimeout`, and `info` are ignored if the invoker withdraws an earlier preemption request, and `NegotiateRecords` will then be an empty list.

If a Resource Manager receives a `Negotiate` command, it will retrieve the primary owners for all specified resources by `Fcm::GetReservationStatus`. To all primary clients it sends a `PreemptionRequest` command, specifying all resources for which the client is the primary client. It sets a timer with the specified timeout value, and awaits all responses during the timeout period. It returns the call from the contender after all expected replies have been received, or shortly after the timeout, compiling the resource negotiate records for all resources.

If the current primary client of a resource is a system client, a negotiation for that resource will always be rejected by a Resource Manager. Note that negotiation is not possible for acquiring secondary access rights.

Error codes

- `ResourceManager::EREJECTED` – the negotiation did not succeed for at least one client

<Client>::PreemptionRequest

Prototype

```
Status <Client>::PreemptionRequest(
      in boolean request, in sequence<SEID> resources,
      in SEID contender, in long preemptionTimeout,
      in ClientRole role, in wstring<50> inInfo,
      out wstring<50> outInfo)
```

Parameters

- `request` – if `True`, this is a preemption request; otherwise a preemption withdrawal
- `resources` – the resources this client has reserved as primary, and for which the negotiation takes place. The safe parameter size limit is 20 `SEID` values.
- `contender` – the contender for the resources
- `preemptionTimeout` – the maximum period in seconds the client is expected to accept or reject the preemption request
- `role` – the user or system role of the contender
- `inInfo` – an information string submitted by the contender
- `outInfo` – an information string submitted by the current client

Description

This message is sent by a Resource Manager to the primary client of the `resources`. The client should process the message within the specified timeout period for a preemption request (and immediately for a preemption withdrawal). Before the timeout expires, the client should reply. `inInfo` is an additional information string that may be supplied by the contender. If a client is too late with its response, the reply will be ignored (`ResourceManager::Negotiate` will return "timeout has expired").

If the client accepts the preemption request, the method returns successfully. Otherwise it returns with error code `EREJECTED`. `outInfo` may carry extra information from the client to the contender. As a guideline, a request for resources for which the client is not the primary client should be rejected by it, and unexpected withdrawals should be ignored. These situations will normally not occur, but cannot be avoided when several reservations and releases occur at the same time.

`preemptionTimeout`, `inInfo`, and `outInfo` can be ignored if this is a preemption withdrawal. The operation code used to send the `PreemptionRequest` message is (one of the operation codes) supplied by the client via `ResourceManager::Reserve`.

Error codes

- `ResourceManager::EREJECTED` – the negotiation was not accepted for at least one client
- `ResourceManager::ENOT_SUPPORTED` – the application does not support preemption requests

ResourceManager::ScheduleAction

Prototype

```
Status ResourceManager::ScheduleAction(
      in SEID controllerApplicationId,
```

```
in SEID triggerId,
in OperationCode awakeNotification,
in sequence<Command> startCommandsList,
in sequence<Command> stopCommandsList,
in sequence<SAConnection> connectionList,
in DateTime startTime,
in DateTime stopTime,
in SAPeriod periodicity,
in sequence<HUID> involvedFcmList,
in wstring<50> userInfo,
out long index)
```

Parameters

- `controllerApplicationId` – *optional SEID* of the application that may be awoken during a Scheduled Action. For this parameter, the SEID of the invoked Resource Manager means that no controller application is used during this Scheduled Action
- `triggerId` – *optional* SEID of the DCM/FCM that may generate the Scheduled Action triggering signal. For this parameter, the SEID of the invoked Resource Manager means that no trigger is used during this Scheduled Action
- `awakeNotification` – the operation code of the controller application's awake notification message (to be invoked by the Action Scheduler). This parameter is ignored if `controllerApplicationId` is absent.
- `startCommandsList, stopCommandsList` – commands to be executed during the Scheduled Action. The commands are listed by order of execution. The safe parameter size limit for each parameter is 2 Kbytes.
- `connectionList` – list of connections that have to be performed during the Scheduled Action. The safe parameter size limit is 20 `SAConnection` values.
- `startTime, stopTime` – date and time information for the Scheduled Action
- `periodicity` – indicates the periodicity of the Scheduled Action. If a periodicity is specified, the Scheduled Action starts and stops every day or every week at time indicated in `startTime`
- `involvedFcmList` – list of the FCMs to be reserved at the start time of the Scheduled Action. The safe parameter size limit is 10 `HUID` values.
- `userInfo` – string field containing the reservation reason
- `index` – references the Scheduled Action

Description

Defines a Scheduled Action. This data is given by the invoking application to the Action Scheduler. It is checked whether the Scheduled Action is valid (all involved FCMs exist and are present at scheduling time, FCMs can accept the commands, there is enough bandwidth to perform connections, etc...).

Each time a Scheduled Action is created, it is referenced with an index that is returned by the Action Scheduler in charge of this new Scheduled Action.

Error codes

- `ResourceManager::EMISSING_RES` – at least one of the involved FCMs are not present at scheduling time
- `ResourceManager::ESCHED_OVERLAP` – at least one of the involved FCMs has already scheduled another Scheduled Action at the specified time
- `ResourceManager::ECOMM_CHECK` – at least one of the involved FCMs cannot deal with the specified commands
- `ResourceManager::EINSUFF_BANDWIDTH` – not enough network resources
- `ResourceManager::ECONT_SEID` – `controllerApplicationId` is invalid

- `ResourceManager::ETRIGG_SEID` – `triggerId` is invalid
- `ResourceManager::ETIME` – incorrect time information (e.g., incorrect format, invalid or "ignored" values in `DateTime`, start time > stop time, stop time < current time)
- `ResourceManager::EINV_PLUG` – at least one of the plug connections is invalid

ResourceManager::UnscheduleAction

Prototype

```
Status ResourceManager::UnscheduleAction(in long index)
```

Parameter

- `index` – reference to the Scheduled Action

Description

Unschedules the Scheduled Action which has the given index. This `UnscheduledAction` is sent to the Action Scheduler that maintains the Scheduled Action that has to be cancelled.

If this method is called while the Scheduled Action is executing, the execution is aborted, and an `ErroneousScheduledAction` event is posted. The involved DCMs shall be sent `UnscheduleReservation` messages by the Action Scheduler.

Error codes

- `ResourceManager::EINV_INDEX` – the `index` is invalid

ResourceManager::GetLocalScheduledActions

Prototype

```
Status ResourceManager::GetLocalScheduledActions(
      out sequence<long> indexList)
```

Parameters

- `indexList` – list of Scheduled Action indexes. The safe parameter size limit is 20 `long` values.

Description

Gets the index list of all Scheduled Actions (if any) undertaken by a Resource Manager. The list is empty if no Scheduled Actions have been undertaken by this Resource Manager.

ResourceManager::GetScheduledActionData

Prototype

```
Status ResourceManager::GetScheduledActionData(
      in long index,
      out boolean valid,
      out SEID controllerApplicationId,
      out SEID triggerId,
      out OperationCode awakeNotification,
      out sequence<Command> startCommandsList,
      out sequence<Command> stopCommandsList,
      out sequence<SAConnection> connectionList,
```

```
      out DateTime startTime,
      out DateTime stopTime,
      out SAPeriod periodicity,
      out sequence<HUID> involvedFcmList,
      out wstring<50> userInfo)
```

Parameters

- `index` – references the Scheduled Action for which the data are to be retrieved
- `valid` – `True` if the Scheduled Action is currently valid, `False` if it is invalid, i.e., it is still missing one or more resources required for (future) execution
- `controllerApplicationId` – *optional SEID* of the application that may be awoken during a Scheduled Action. For this parameter, the SEID of the invoked Resource Manager means that no controller application is used during this Scheduled Action
- `triggerId` – *optional* SEID of the DCM/FCM that may generate the Scheduled Action triggering signal. For this parameter, the SEID of the invoked Resource Manager means that no trigger is used during this Scheduled Action
- `awakeNotification` – the operation code of the controller application's awake notification message (to be invoked by the Action Scheduler)
- `startCommandsList, stopCommandsList` – commands to be executed during the Scheduled Action; the commands are listed by order of execution. The safe parameter size limit for each parameter is 2 Kbytes.
- `connectionList` – list of connections that have to be performed during this Scheduled Action. The safe parameter size limit is 20 `SAConnection` values.
- `startTime, stopTime` – date and time information for the Scheduled Action
- `periodicity` – indicates the periodicity of the Scheduled Action; if a periodicity is specified, it indicates that the Scheduled Action starts and stops every day or every week at time indicated in `startTime`
- `involvedFcmList` – list of the involved FCMs. The safe parameter size limit is 10 `HUID` values.
- `userInfo` – string field containing the reservation reason

Description

Gets Scheduled Action data corresponding to a given index. Note that `startTime` and `stopTime` remains the same as those registered by `ResourceManager::ScheduleAction`, even though the corresponding Scheduled Action with 'periodicity' has once (or more) been executed

Error codes

- `ResourceManager::EINV_INDEX` – `index` is invalid

ResourceManager::TriggerNotification

Prototype

```
Status ResourceManager::TriggerNotification(
      in long index, in boolean startStop)
```

Parameters

- `index` – local reference to the Scheduled Action
- `startStop` – indicates the type of the triggering notification: if `True`, it is a start indication; otherwise, it is a stop indication.

Description

The `TriggerNotification` is issued by the trigger, e.g. an FCM or DCM, as a result of the set

up (subscription or otherwise) by the invoking application (the Action Scheduler identification and the index of the Scheduled Action should have been passed to the trigger).

Error codes

- `ResourceManager::EINV_INDEX` – `index` is invalid

<Client>::AwakeNotification

Prototype

```
Status <Client>::AwakeNotification()
```

Description

The awake notification is a general purpose message of an application to indicate to it that the caller has accomplished whatever the application was waiting for ("whatever" is proprietary between caller and application). This method is used for scheduled actions.

The operation code used to send the `AwakeNotification` message is indicated by the client via `ResourceManager::ScheduleAction`.

ResourceManager::GetScheduledConnections

Prototype

```
Status ResourceManager::GetScheduledConnections(
      in DateTime startTime,
      in DateTime stopTime,
      in SAPeriod periodicity,
      out sequence<RMConnection> connections)
```

Parameters

- `startTime, stopTime` – date and time information for the Scheduled Action
- `periodicity` – periodicity of the Scheduled Action
- `connections` – list of all connections active between start and stop time. The safe parameter size limit is 100 `RMConnection` values.

Description

Get a list of all connections that will be active between start and end time. For periodical schedules, future intervals also have to be taken into account. The list is empty if no connections match.

5.10.4 Resource Manager Events

InvalidScheduledAction

Prototype

```
void InvalidScheduledAction(
      in SAReference ref, in wstring<50> userInfo)
```

Parameters

- `ref` – reference of the Scheduled Action that is not valid
- `userInfo` – reservation reason string of the Scheduled Action

Description

This event can be issued between scheduling time and stop time of the Scheduled Action. Depending on the software element that outputs this event, it indicates that either FCMs, network resources, or Action Scheduler are not all gathered for a successful execution of the Scheduled Action. Issuing this event does not involve the deletion of the Scheduled Action.

An Action Scheduler issues this event when a Scheduled Action cannot be completed in the current situation (a resource disappears, not enough bandwidth available ...).

A DCM involved in a particular Scheduled Action (via `Dcm::ScheduleReservation`), issues this event when its Action Scheduler is missing in the current situation. Unlike an Action Scheduler, a DCM shall delete its related Restricted Scheduled Action data.

The event indicates the reference of the Scheduled Action, and its reservation reason for user information purposes.

AbortedScheduledAction

Prototype

```
void AbortedScheduledAction(
      in SAReference ref, in wstring<50> userInfo)
```

Parameters

- `ref` – reference of the Scheduled Action that is aborted
- `userInfo` – reservation reason string of the Scheduled Action

Description

This event is issued by the Action Scheduler when the Scheduled Action is not executing in the following three cases:

- If the control application or trigger (if any) disappears.
- If the required resources (FCMs, network resources) are still not present, at or after the stop time of a non-triggered (invalid) Scheduled Action – i.e., the Scheduled Action remained invalid.
- If the required resources for a Scheduled Action become available again, but the restart of the Scheduled Action fails.

Issuing this event means that the specified Scheduled Action data has been deleted by the Resource Manager. Also the related Restricted Scheduled Action data is removed from the involved DCMs by the Resource Manager (via `Dcm::UnscheduleReservation`).

This event indicates the reservation reason of the Scheduled Action for user information purposes.

ErroneousScheduledAction

Prototype

```
void ErroneousScheduledAction(
      in SAReference ref, in wstring<50> userInfo)
```

Parameters

- `ref` – reference of the Scheduled Action that has encountered an error during execution
- `userInfo` – reservation reason string of the Scheduled Action

Description

This event is issued during execution of the Scheduled Action if one of the following errors occur:

- start/stop command or connection fails
- reservation fails
- controller application or trigger disappears
- FCM disappears or is preempted

The Scheduled Action is aborted, unless it is controlled by an application. In that case the application may decide to unschedule (and thereby abort) it.

5.10.5 Bandwidth Checking Protocol

To get a snapshot of all scheduled isochronous connections distributed on the network for a specified period of time, an Action Scheduler has to query all other Action Schedulers to obtain their relevant schedules. To enable this, the inter-Action Scheduler method `ResourceManager::GetScheduledConnections` is defined. The Resource Manager that acts as an Action Scheduler will invoke this method on all other Resource Managers. It adds the required bandwidth of all Action Schedulers (including itself) to its own required bandwidth for overlapping time periods in the future. If this calculation exceeds the available bandwidth, the remaining network bandwidth is assumed unavailable for the newly planned scheduled action.

Notes:

- In principle, when several Action Schedulers are checking concurrently, a scheduled action may unnecessarily be rejected. However, this scheme suffices for home networks, where only a few concurrent schedules will occur.)
- For calculation of the actual bandwidth needed see the section 5.9.5.4.1.

In the example below a scheduled action is going to be registered at *RM1* as Schedule B. For bandwidth checking *RM1* contacts the other Action Schedulers of the network (which are *RMa* and *RMb*) to get schedules that are concurrent (i.e. overlapping) with the schedule *RM1* is working for. The request contains the scheduling time information, so that *RMa* and *RMb* return only the information about those scheduled connections, which are concurrent. Comparing this information with its own schedules, *RM1* becomes able to compute whether the remaining network resources are enough to ensure the completion of all scheduled actions programmed in the future.

Inter-resource manager communication for checking bandwidth at scheduling time is depicted below:

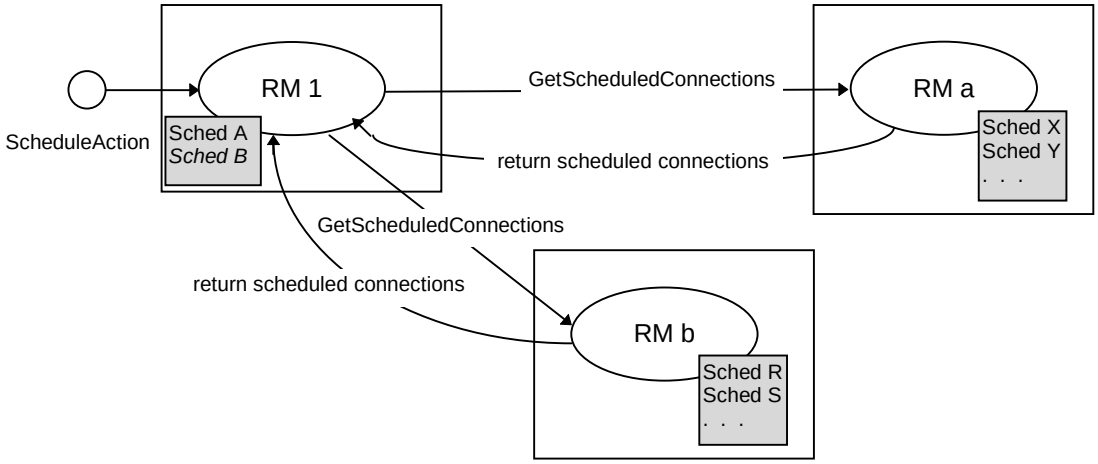

Figure 34. Resource Managers and Bandwidth Checks

Bandwidth checking reference algorithm. The algorithm below describes a possible implementation guideline, the actual implementation is of course proprietary. The algorithm does not take into account different link speeds in the network:

There is a new schedule A which includes the start time s1, the end time e1, the needed bandwidth B1 and the needed number of channels C1. The problem is to determine whether this new schedule can be added to the action scheduler.

Collect registered schedules with an end time later than s1 and with a start time earlier than e1 and make a schedule table of schedules that are affected by the new schedule A. An example of a schedule table is:

Start time	End time	Bandwidth needed	No. of channels needed
Sa	Ea	6Mbps	1
Sb	Eb	6Mbps	2
Sc	Ec	5Mbps	1
Sd	Ed	5Mbps	1
Se	Ee	5Mbps	2

Make a bandwidth table which includes as check point times: s1, the later start times than s1 and the earlier end times than e1 from the schedule table. A bandwidth table then looks like:

Check point time	Bandwidth needed	No. of channels needed
C1	12Mbps	2
C2	15Mbps	4
C3	10Mbps	3

The bandwidth table is built by calculating for each check point the bandwidth and number of channels from the schedule table. The calculation steps are:

- Find all schedules containing the check point from the schedule table.
- Make the summations (or subtractions) of bandwidth and number of channels for all the found schedules.
- Put the summations to the bandwidth and channel slot of the bandwidth table.

Obtain maximum bandwidth (Bmax) and maximum number of channels (Cmax) from the bandwidth table. Calculate:

Bmax + B1 and Cmax + C1

Then check with the available network bandwidth and channel numbers for overflow caused by the new schedule.

For efficiency reasons the overflow check done at the end could be done during the calculations of step (3).

5.11 Application Module

This section provides the common set of Application Modules commands. Besides the commands described here, an Application Module may also provide the DDI API. When it does, it indicates so by the `ATT_GUI_REQ` attribute in the Registry. Moreover, as holds for all HAVi components, proprietary extensions of this command set are allowed.

5.11.1 Services Provided

Service	Comm Locality Type	Access

Service	Comm Type	Locality	Access
ApplicationModule::GetIcon	M	global	all
ApplicationModule::GetHuid	M	global	all
ApplicationModule::GetHavletCodeUnitProfile	M	global	all
ApplicationModule::GetHavletCodeUnit	M	global	all

5.11.2 Application Module Data Structures

HUID

The HUID is the unique identification of a DCM, FCM or Application Module. The structure for HUIDs is given in section 5.6.2.

5.11.3 Application Module API

ApplicationModule::GetIcon

Prototype

```
Status ApplicationModule::GetIcon(
      out DeviceIcon icon)
```

Parameters

- `icon` visual representation of the application.

Description

Provides a visual representation of the application that can be displayed to the user. `DeviceIcon` is defined in section 5.6.2.

ApplicationModule::GetHuid

Prototype

```
Status ApplicationModule::GetHuid(out HUID appId)
```

Parameters

- `appId` – a HUID of an Application Module

Description

Returns the HUID of the Application Module.

ApplicationModule::GetHavletCodeUnitProfile

Prototype

```
Status ApplicationModule::GetHavletCodeUnitProfile(
      out Version version,
      out long transferSize,
```

```
      out long codeSpace,
      out long workingSpace,
      out long chunkSize)
```

Parameters

- `version` – the lowest version of the HAVi Messaging System required by this havlet.
- `transferSize` – the number of havlet code unit bytes to be transferred
- `codeSpace` – the number of bytes required for the installed havlet code unit (read-only part)
- `workingSpace` – an estimate of the number of bytes required for the working space of the installed havlet code unit (read/write part)
- `chunkSize` – the maximum number of havlet code unit bytes the Application Module can send at a time

Description

Provides the various size parameters needed for determining whether the destination of the havlet code unit (for example, an FAV UI Manager) can install the havlet. This method is only supplied if the Application Module indicates that it supports havlets via the `ATT_GUI_REQ` attribute in the Registry.

Error codes

- `ENOT_IMPLEMENTED` – if the Application Module does not contain a havlet.

ApplicationModule::GetHavletCodeUnit

Prototype

```
Status ApplicationModule::GetHavletCodeUnit(
   in long firstByte,
   in long lastByte,
   out sequence<octet> byteArray)
```

Parameters

- `firstByte` – the number of the first byte of the transferred havlet code unit byte array wanted
- `lastByte` – the number of the last byte of the array wanted
- `byteArray` – the byte array requested (empty if none could be delivered or if invalid `firstByte` and/or `lastByte` values are supplied). The safe parameter size limit 512 bytes (the 1394 asynchronous packet size for bus speeds of 100 Mbps).

Description

Provides the bytecode that can used to install and execute an havlet, a Level 2 application stored in this Application Module. A havlet code unit receiver can request all or some of the bytes of the havlet code unit from the Application Module. It should first use `ApplicationModule::GetHavletCodeUnitProfile` to determine if it is capable of retrieving and installing the code unit. `firstByte` and `lastByte` should indicate subsequent parts of the code unit to be transferred. The first byte of the code unit is number 1; the last byte is the value of `transferSize`. The amount of bytes requested (`lastByte` - `firstByte` + 1) shall not exceed the value of `chunkSize`.

The format of the bytecode and the way it should be handled by an FAV are described in section 7.4.3. This method is only supplied if the Application Module indicates that it supports havlets via the `ATT_GUI_REQ` attribute in the Registry.

Error codes

- ENOT_IMPLEMENTED – if the Application Module does not contain a havlet.
- EINVALID_PARAMETER – the values of firstByte and/or lastByte are invalid

5.12 APIs for Data Driven Interaction

This section describes in detail the DDI-related data structures and operations that have been described more generally in section 4 *Data Driven Interaction*. This section relies on the concepts and terminology defined in section 4.

The data structures described here represent a set of DDI elements that can be extracted by a DDI Controller from a DDI Target and presented by the controller to a human user on a display screen. The controller can then allow the user to manipulate and indicate selection of the displayed elements and thereby cause the controller to send corresponding user actions to the target. These user actions are represented by data structures (associated with the DdiTarget::UserAction operation) which are also defined in this section. It is also possible for the target to notify the controller of changes in the target by sending (using a message back operation) a representation of the change as a list of affected element IDs. Element data structures can be referenced using unique element IDs. The elements whose selection can give rise to user actions are called interactive elements. Elements that are not interactive may be used by the target to provide information about the target to the user where it is not necessary to allow the user to provide input to the target.

The controller and target use the operations defined below to carry on this interaction. The DDI Controller opens an interaction with a target using the DdiTarget::Subscribe operation, interacts with the target using the other DdiTarget:: and <Client>:: (i.e. DDI Controller) operations, then closes the interaction with that particular target using the DdiTarget::Unsubscribe operation. During the interaction:

- DDI elements are obtained by the controller from the target using the GetDdiElement, GetDdiPanel, GetDdiGroup, and GetDdiElementList operations. It is up to the controller to determine how many DDI elements are retrieved and when they are retrieved. The target only knows about what was last pulled from it by the above operations.

- other data structures, DDI content, which can be considered to be logical parts of the DDI element that reference them, may be obtained by the controller using the GetDdiContent operation.

- the controller may send a user action to the target using the UserAction operation.

- as noted above, the target may use the <Client>::NotifyDdiChange message back operation to indicate to the controller that particular elements have changed and that their new values may be obtained from the target using the DdiTarget::GetDdi… operations mentioned above.

A controller may carry on an interaction with more than one target at a time. A target may carry on interactions with more than one controller at a time. In this latter case, the target should coordinate (in an implementation-dependent manner) user actions from and notifications to each controller to ensure that each controller's DDI state is consistent with the current target state.

The DDI version level is supported as described in section 5.13.2 *Version Control API*.

5.12.1 Services Provided

Service	Comm Type	Locality	Access
DdiTarget::Subscribe	M	global	all
DdiTarget::Unsubscribe	M	global	all
DdiTarget::GetDdiElement	M	global	all
DdiTarget::GetDdiPanel	M	global	all
DdiTarget::GetDdiGroup	M	global	all
DdiTarget::GetDdiElementList	M	global	all
DdiTarget::GetDdiContent	M	global	all
DdiTarget::ChangeScope	M	Global	all
DdiTarget::UserAction	M	global	all
<Client>::NotifyDdiChange	MB	global	DDI Target (all)

5.12.2 Presentation Requirements and Recommendations for DDI Controllers

The DDI protocol places a small number of mandatory presentation requirements on a DDI Controller: how a controller must output DDI data to its human user and how the controller must accept input from its user and transform it into DDI data (i.e., actions). The DDI protocol also includes a larger number of recommendations for the controller's user input/output operation. These requirements and recommendations are of two kinds: *general* and *specific*. General requirements and recommendations are given here; specific ones, will be given in the sub-sections describing particular elements, attributes, actions, and operations. In the case of requirements, the words "required" or "must" and their variants will be used. In the case of recommendations, the words "recommended" or "suggested" and their variants will be used.

5.12.2.1 General Presentation Requirements for DDI Controllers

The general presentation requirements placed on a DDI Controller are to:

- Display (using a controller-specific rendering, given its user input/output capabilities) the information contained in the attributes of the DDI elements that the controller has pulled from the target. Note, this requirement is only specified abstractly: the controller *must* give a "best-effort" (relative to its actual capabilities) presentation of the information embodied in the elements; however, no *particular* form or level of rendering is required.

- It is however required that the controller makes it possible for a user to see all DDI elements of a panel, e.g. by using scrolling facilities. It is also required that a controller makes it possible for a user to access all reachable panels (those for which the target supplied panel links).

- Allow the user to manipulate in a controller-specific, best-effort manner any interactive DDI element that the controller has pulled from the target so as to result in the generation of any action (with any attribute values that the action is capable of having) associated with that interactive element. Note, again this requirement is expressed abstractly with respect to the particular user-controller interaction that the controller uses to allow the user to: choose an interactive element, specify the attributes of an action, and indicate that the action should be sent by the controller to the target.

- If a target shows a DDI element to the user, it is required to be the most recent, correct version (not an old, obsolete version). Therefore the controller has to pull (or more often, pull again) from the target all elements whose identifiers are listed in change reports from the target if the user needs the information contained in those elements.

- All elements (except the `DDI_TEXT` element which already contains a text string) have a mandatory label attribute that is defined to be a text string of length between zero and 16 characters. These labels have to be shown by the controller (minimal controllers can use scrolling if needed).

5.12.2.2 General Presentation Recommendations for DDI Controllers

Presentation recommendations for DDI Controllers allow for two types of scaling: panel scaling and element scaling. Both types of scaling are explained below.

5.12.2.2.1 Panel Scaling

A panel element is intended to represent an entire screen's worth of DDI elements. Within the panel can be zero or more groups, each of which may contain individual elements or other groups. In addition, a panel can have standalone elements that are not part of any group.

For controllers showing several panels on their screen, the navigation between these panels is proprietary (there are however recommendations with respect to usage of panel links).

A controller can present a given panel in one of three styles: full-capability, intermediate, and basic.

- If the controller's panel is displayed on the screen as required by all elements supplied by the target then the controller is said to be operating in "full-capability".

- If the controller cannot display the entire panel as given it can then drop down to the next style of "intermediate". Suggestions on how to scale down would be represented by the target in its placement of elements within groups, groups within groups, and groups within a panel. The intermediate controller could then display one group at a time and allow the user to locally navigate between groups by displaying next and previous groups.

- The order in which the elements and groups of a panel should be shown is determined by their order inside the panel/group. It is recommended that controllers make sure that the user knows the hierarchical level of the shown group within the panel. E.g. by showing (as much as possible) the labels of the other elements or groups.

- The last style, "basic", would be used if the controller can only display a few or one DDI element at a time. The basic controller could then display a set of elements at a time and allow the user to locally navigate between elements.

- The order in which the elements should be shown is determined by their order inside the panel/group. It is recommended that controllers make sure that the user knows the hierarchical level of the shown elements within the panel. E.g. by showing (as much as possible) the labels of the other elements or groups.

Which of the above styles the controller chooses depends on its capability and also whether the screen is concurrently used for other purposes (e.g. video rendering).

Note that the capabilities that the target assumes for element sizes and position might clash with the controller's capabilities for these attributes. This can still make it impossible for a controller that is in principle "full-capability", to show all the elements of the panel together.

5.12.2.2.2 Element Scaling

As given in section 5.12.2.1 *General Presentation Requirements for DDI Controllers*, a controller is required to display information contained in DDI elements. However, the controller has the freedom to choose the particular rendering and user interaction operation. Especially for non-organisational elements, the controller can choose between different rendering possibilities.

DDI element optional attributes are intended to be suggestions to the controller on how to organize and present the target's panels, groups, and elements to the user.

A controller has to take into account the DDI element mandatory attributes, but in case the controller does not choose to represent graphical attributes (e.g. text-only controllers), it can neglect even specific (graphical) mandatory attributes. These mandatory graphical attributes that might be neglected are: aspect ratio, height, width, and bitmaps. In this case, the label attribute is used to represent an element to the user.

Ideally DDI elements should be positioned on the panel or group according to the (optional) position attribute supplied by the target. If this is not possible (not supplied by target, basic panel style chosen, etc.), the controller is recommended to render the elements on the screen in the order supplied by the target in either horizontal or vertical direction.

It is recommended that, if targets supply position attributes, they do this for all elements of the group/panel. Note that targets should not rely on the precise position of the elements that they specify.

The above guidelines allow controllers to use rather different styles of representing DDI elements by ignoring or adapting certain types of attributes. It is recommended that a controller uses the same style for all the elements of the same panel or group.

5.12.3 DDI Data Structures Overview

The DDI elements are the data structures that a DDI Target can use to define how the target should be represented on a display screen by a DDI Controller.

An element is represented as a `struct` of a particular type. Attributes for an element are represented by the fields of the `struct`. There are two kinds of attributes: *mandatory* and *optional*. Each mandatory attribute is represented by an explicit field, starting with the first field of the element. The mandatory attributes of a DDI element must be included in the element. Optional attributes are contained in a variable length list (possibly empty) contained in the last explicit field of the element. The optional attributes need not be present but if included are intended to give the controller suggestions on displaying this particular DDI element

All DDI elements have the mandatory attributes height and width indicating the size (in pixels) that the element should occupy on a graphics capable controller. The optional attribute, position, is used to place the element on the panel or group. The height, width and position of a panel are specified relative to a safety area of 640 x 480 pixels on which any full capability controller can surely display a panel. A panel is defined to be two-dimensional with the upper left corner being defined as (x = 0, y = 0). All positions given for groups or standalone elements directly contained in a panel's list of elements are relative to this origin. Similarly, the elements within a group have position values based on the group's upper-, left-most position being (x = 0, y = 0). The x-coordinate is positive incrementing from left to right. The y-coordinate is positive incrementing from top to bottom.

The "navigation" from one panel to another panel is done by the controller getting the next panel as contained in the `panelLink` attribute of the DDI element `DdiPanelLink`. The user selecting the `DdiPanelLink` element does not result in any user action that retrieves the panel. Instead, the controller itself has to requests the panel in the `DdiPanelLink` by using the operations `GetDdiPanel`, `GetDdiElement` or `GetDdiElementList`.

Panels and groups can have background colors, pictures or patterns. A pattern could take the form of a tiled sequence of icons, this would allow for textured appearances

Panels and groups can also have audio/video content streamed in real-time. When an `audioVideo` attribute is given in DDI data, a DDI Controller can use the audio video stream for background video, etc. If the `audioVideo` attribute is valid, a DDI Controller should consider `audioVideo` as higher priority to "background picture", "background pattern" and "background color".

Furthermore, panels and groups can have audio content streamed in real-time. When an `Audio` attribute is given in DDI data, a DDI Controller can use the audio stream for background music, etc.

For DDI elements having the (optional) sound attribute a controller should give that sound priority over the (audio part) of the background audio/video or audio content (if present).

5.12.4 Basic DDI Types

The data types described here are used in a number of other DDI data structures and are not necessarily associated with a particular element type or attribute type.

DdiElementType

Definition

```
typedef ushort DdiElementType;
```

Description

The particular `ushort` values for the DDI element types are specified in Annex 11.15. The data type representing each DDI element type is given in section 5.12.8 *Individual DDI Elements*.

DdiElementId

Definition

```
struct DdiElementId {
      DdiElementType      ddiElementType;
```

```
      ushort              ddiElementHandle;
};
```

Description

`DdiElementId` is used for the identification of any of the DDI elements specified by this section. The `ddiElementHandle` field is defined by the DDI Target. This field is not meant to be interpreted and should not be modified by the DDI Controller.

A target always returns the same element for a given `DdiElementId`.

Every DDI element includes a mandatory attribute (in the element's first field) of type `DdiElementId` which is an identifier for that DDI element. See section 5.12.8 *Individual DDI Elements*.

DdiContentId

Definition

```
struct DdiContentId {
      DdiContentType      ddiContentType;
      ushort              ddiContentHandle;
      uint                ddiContentSize;
};
```

Description

`DdiContentId` is used for the identification of any DDI content data structure – see section 5.12.5 *DDI Content Formats* for `DdiContentType`. The `ddiContentSize` field contains the total size of the content data in bytes. The target may use this size to make appropriate preparations in anticipation of receiving the actual content data. If the content is empty, `ddiContentSize` should be equal to 0. The `ddiContentHandle` field is defined by the DDI Target. This field is not meant to be interpreted and should not be modified by the DDI Controller. The `DdiContentId` is returned to the target when the controller requests the value of the content data (via `GetDdiContent`).

A target always returns the same piece of content for a given `DdiContentId`.

DdiElementIdList and DdiElementList

Definition

```
typedef sequence<DdiElementId>   DdiElementIdList;
typedef sequence<DdiElement>     DdiElementList;
```

Description

The above types are used for referring to lists of DDI element IDs and elements.

DdiColor

Definition

```
typedef octet DdiColor[4];
```

Description

The order of the octets in `DdiColor` is: alpha, R, G, B (true color). The alpha field shall specify the transparency of the pixel with respect to the background. A value of 0 indicates the pixel is fully transparent, a value of 0xff indicates that the pixel is opaque.

Bitmap and Sound

Definition

```
typedef sequence<octet> Bitmap;
typedef sequence<octet> Sound;
```

Description

The format and size of a `Bitmap` will be determined by the data structure within which it is contained – see section 5.12.5 *DDI Content Formats*. The width and height of a bitmap should be no larger than the specified width and height of the DDI element that it is (directly or indirectly) contained in.

The format and size of a `Sound` will be determined by the data structure within which it is contained – see section 5.12.5 *DDI Content Formats*.

A "bitmap" is a single image and a "sound" is audio data identified by `DdiContentId`. A bitmap and/or sound can optionally be part of certain DDI elements and is intended to be a small amount of data. For example, a sound may be used to attach short audio output data items to button elements for presentation during button presses and releases. Note, bitmaps and sounds are distinct from media streams being played back in real-time, usually over an extended period of time – see `AudioVideo` or `Audio` below. The term "audio/video" will be used when media stream references are required.

AudioVideo

Definition

```
struct AudioVideo {
      SEID    dcm;
      uint    handle;
};
```

Description

`AudioVideo` is used to represent a media stream source on the target associated with the specified `handle`. The handle is specified by the target and could also (but does not need to) be used in a content icon list. When used in an attribute by the target, `AudioVideo` allows the controller to setup audio/video media streams using the `DCM::SelectContent`. If a DDI element has audio audio/video stream content then it must be supported by some DCM (specified by `dcm`). If the controller can accept and support media streams then the controller should use `SelectContent` with the `contentType` field set to `AV`.

There is no mandatory audio/video stream format in HAVi DDI.

When an `audioVideo` attribute is given in DDI data, a DDI Controller can use the audio video stream for background video, etc.

Audio

Definition

```
struct Audio {
      SEID    dcm;
      uint    handle;
};
```

Description

`Audio` is used to represent a audio media stream source on the target associated with the specified `handle`. The handle is specified by the target and could also (but does not need to) be used in a content icon list. When used in an attribute by the target, `Audio` allows the controller to setup audio media streams using the `DCM::SelectContent`. If a DDI element has audio stream content then it must be supported by some DCM (specified by `dcm`). If the controller can accept and support media streams then the controller should use `SelectContent` with the `contentType` field set to `Audio`.

There is no mandatory audio stream format in HAVi DDI.

When an `Audio` attribute is given in DDI data, a DDI Controller can use the audio stream for background music, etc.

Label

Definition

```
typedef wstring<16> Label;
```

Description

A `Label` is used within DDI elements from the target to hold textual data meant to be ultimately interpreted by the human user with which the controller is communicating. It is not intended for the controller to interpret the contents of a `Label` or to modify it in a way that would change its meaning to the human user.

NotificationScope

Definition

```
enum NotificationScope {CURRENT, GLOBAL, ADD};
```

Description

The `NotificationScope` type is used as a parameter of `DdiTarget::Subscribe` and `DdiTarget::ChangeScope` that indicates whether the `NotificationScope` should generate DDI change reports only for elements within the current panel (= `CURRENT`), for elements within all available panels (= `GLOBAL`) or for elements within the set of panels retrieved since the last call to `DdiTarget::Subscribe` or `DdiTarget::ChangeScope` (= `ADD`). The current panel is always part of the notification scope.

Interactivity

Definition

```
enum Interactivity {ENABLED, DISABLED};
```

Description

Interactive DDI elements include a mandatory interactivity attribute that indicates whether the element can be used in an interactive (ENABLED) way resulting in defined user actions or in a non-interactive (DISABLED) way, not resulting in user actions. For "normally" interactive DDI elements, Interactivity = DISABLED means that the DDI element is still visible, but for instance grayed-out (this is up to the controller). The "normally" interactive elements are: DdiToggle, DdiButton, DdiBasicButton, DdiSetRange, DdiEntry, DdiChoice.

InformTarget

Definition

```
enum InformTarget {INCREMENTALLY, COMPLETE};
```

Definition

InformTarget indicates whether:

- each change to an interactive element, e.g., a DDI_ENTRY being used for text entry, immediately results in the controller sending an associated user action (InformTarget = INCREMENTALLY);
- or, the controller waits until the user indicates in a controller-dependent manner that user input is finished and only then sends the user action (InformTarget = COMPLETE).

DDI elements (like DdiEntry) with the informTarget attribute set to COMPLETE require the controller to send a user action back to the target only when the user has finished entering the information. When and how the controller knows when to send a user action is determined by the controller. It could be an ENTER button on the remote, typing the ENTER key on a keyboard, etc.

DDI elements (like DdiSetRange) with the informTarget attribute set to INCREMENTALLY require the controller to send the user action back to the target immediately when the user updates the information. Before possible change reports from the target due to this user action arrives back at the controller, the user may already have done more updates. This kind of "type ahead" can be allowed by the controller, but in the case that the user's input has to be ignored it is also the controller's responsibility to indicate this to the user. In the case in which the user's input does not need to be ignored, the controller can send it as a whole to the target as a next user action. So, for DdiSetRange this means that even with informTarget=INCREMENTALLY, a user action can indicate a change larger than stepValue.

Pattern

Definition

```
struct Pattern {
   ushort        height;
   ushort        width;
   DdiContentId  patternBitmapId;
};
```

Description

The controller can present groups of elements as small opaque blocks on the screen, and can either make the panel background opaque or transparent. The controller has the freedom to display both standalone elements and groups in any manner it sees presentable.

The PNG bitmap format, which includes gamma information, can be used to define the characteristics of the screen. In this way colors and brightness levels will appear correct whether the bitmap is displayed on a TV screen or other display device.

Fontsize

Definition

```
enum Fontsize {SMALL, MEDIUM, LARGE};
```

Description

DDI elements can have optional `fontSize` suggestions. In case of full capability devices, as a guideline, `MEDIUM` size is specified as 16x32 pixels (two-byte codes such as kanji code are 32x32), `SMALL` size is specified as 12x24 pixels (two-byte codes are 24x24) and `LARGE` size is specified as 20x40 pixels (two-byte codes are 40x40). `MEDIUM` is the default font size when no `fontSize` is specified. In any case it is required that `SMALL` characters are not larger than `MEDIUM` and `MEDIUM` characters are not larger than `LARGE`.

Position

Definition

```
struct Position {
   ushort xPosition;
   ushort yPosition;
};
```

Description

The maximum value of `Position` components correspond to the safety area size as specified in `DdiPanel`.

DDI elements can overlap by defining their positions to be at an overlapping relative location and using transparent bitmaps attributes. DDI elements should not completely hide each other. The order of `DdiElement`'s in the panel or group also indicates the back to front order. If the position attribute is not specified or if the controller does not handle position or size information, DDI elements should be rendered in a non-overlapping way.

SafetyAreaPosition

Definition

```
enum HorizontalPosition {LEFT, CENTER, RIGHT};
enum VerticalPosition {TOP, MIDDLE, BOTTOM};

struct SafetyAreaPosition {
     HorizontalPosition horizontalPosition;
     VerticalPosition   verticalPosition;
};
```

Description

The desired position of the safety area on the physical display of the controller can be specified using the SafetyAreaPosition attribute.

FocusNavigation

Definition

```
struct FocusNavigation {
      DdiElementId up;   DdiElementId down;
      DdiElementId left; DdiElementId right;
};
```

Description

Targets can indicate how a controller may allow the user to navigate between elements using the FocusNavigation attribute:

DDI elements (not panels and groups) can have optional "focus change" suggestions, which the controller can use to move the focus on screen. The basic set of directions is "up, down, left, right", but the controller is free to have its own input device which can map to these concepts (it does not need to have an IR remote with 4 buttons). Groups will not have navigation suggestions; the only "suggestion" is the order of the group ID's in the panel list. The controller navigates between groups in this order (or some other order if it has a reason to deviate from this).

The DdiElementId's in the FocusNavigation structure should point to DDI elements in the same panel as the DDI element which contains the FocusNavigation structure. It is allowed for the DdiElementId's to point to the containing DDI element itself.

DdiTitle

Definition

```
typedef DdiContentId DdiTitle;
```

Description

DdiTitle is a bitmap used by a panel or group to indicate its purpose. The title must be no greater than 10% of the size of the panel or group.

DdiContentType

```
enum DdiContentType { UNICODE, PNG, HAVi_RAW_BITMAP, AIFF_C,
JPEG };
```

DdiContent

Definition

```
union DdiContent switch (DdiContentType) {
   case UNICODE:         wstring unicodeText;
   case PNG:             Bitmap  pngBitmap;
   case HAVi_RAW_BITMAP: Bitmap  rawBitmap;
   case AIFF_C:          Sound   AIFF_C_sound;
   case JPEG:            Bitmap  jpegBitmap;
```

```
};
```

Description

DDI data can include DDI content data: text data, bitmap (image) data and sound data, and indicate audio/video stream data. DDI content data formats are divided into two distinct classes: mandatory formats and optional formats. The mandatory formats for DDI content data are specified below, and all DDI Controllers shall at least support the mandatory text format.

5.12.5 DDI Content Formats

5.12.5.1 Text data

All text defined in the DDI element will be in UNICODE which is specified in the reference [12].

5.12.5.2 Image data

Both PNG , JPEG and HAVi raw bitmap are mandatory DDI content image formats for graphics-capable controllers.

- PNG (Portable Network Graphics)

  Details of this format are given in the PNG specification [10]. DDI Controllers are not required to implement the full PNG specification, there are the following limitations:

Limitations : (A) the PNG image file is restricted to a single image
(B) color type = 3 (index color, max. 8 bit color)
(C) bit depth = 1,2,4,8 (max. 8 bit color)

- JPEG

  Details of this format are given in the JPEG International Standard, using the JPEG File Interchange Format (JFIF)[18]. DDI Controllers are not required to implement the full JPEG specification, there is the following limitation:

  (A) only coding using sequential DCT-based mode is required to be supported

  Note that a JPEG image may include a thumbnail, but the HAVi platform is not required to display it.

- HAVi raw bitmap

  HAVi raw bitmap, which is defined by HAVi, is also a mandatory DDI content image format. The HAVi raw bitmap format described below uses the same style IHDR, PLTE chunk formats described in the PNG specification. An alternative RdAT chunk defined by HAVi is used instead of the standard IDAT chunk.

  A. File signature:
  The first eight bytes of a HAVi raw bitmap file always contain the following values:

(decimal)	137	72	65	86	105	13	10	26
(hexadecimal)	89	48	41	56	69	0d	0a	1a

(ASCII C notation) \211 H A V i \r \n \032

B. IHDR:
The IHDR chunk must appear first in the bitmap data. It contains:

Width:	4 bytes	
Height:	4 bytes	
Bit depth:	1 byte	(value : 8)
Color type:	1 byte	(value : 3, index color, max. 256 color)
Compression method:	1 byte	(value : 255, no compression)
Filter method:	1 byte	(value : 255, no information)
Interlace method:	1 byte	(using PNG method)

C. PLTE:
The PLTE chunk contains from 1 to 256 palette entries, and each entry has a three-byte series of the form: Red(1 byte), Green(1 byte), Blue(1 byte).
The first entry in PLTE is referenced by pixel value 0, the second by pixel value 1, etc. the PLTE chunk may have fewer entries than the bit depth. In that case, any out-of-range pixel value that found in the RdAT chunk is an error.

D. RdAT:
The RdAT chunk contains the actual image data. To create this data
image scan lines are represented as described in PNG Image layout. The layout and total size of this raw data are determined by the fields of IHDR in the PNG specification.

E. IEND:
The IEND chunk shall appear last, and it's data field is empty. This IEND chunk shows the end of the HAVi raw bitmap.

5.12.5.3 Sound data

AIFF-C is a mandatory HAVi sound format (AIFF-C is specified in reference [11]). DDI Controllers are not required to implement the full AIFF-C specification, there are the following limitations:

Limitations: (A) sample size = 8 bit
(B) sample rate = 22.050 kHz

5.12.6 DDI Mandatory Attributes

The mandatory DDI attributes (rows) for each DDI element type (columns) are shown in the following table.

Table 10. Mandatory Attributes of DDI Elements

Mandatory Attributes	DdiPanel	DdiGroup	DdiPanelLink	DdiButton	DdiBasicButton	DdiToggle	DdiAnimation	DdiShowRange	DdiSetRange	DdiEntry	DdiChoice	DdiText	DdiStatus	DdiIcon
elemId	X	X	X	X	X	X	X	X	X	X	X	X	X	X
label *elem*Name	X	X	X		X	X	X	X	X	X	X		X	X

height, width	X	X	X	X	X	X	X	X	X	X	X	X	X	X
aspectRatio	X													
elements	X	X												
interactivity			X	X	X	X	X	X	X	X	X	X		X
linkBitmapDdiContentId			X											
panelLink			X											
pressedLabel				X										
releasedLabel				X										
numOffStates						X								
toggleStates						X								
state						X								
repetition							X							
speed							X							
animations							X							
orientation								X	X					
valueRange								X	X					
stepValue								X	X					
valueSet								X	X					
informTarget									X	X				
entryType										X				
qualifier										X				
defaultEntry										X				
maxCharsDigits										X				
choiceType											X			
choiceNumber											X			
wrapType											X			
choiceOrientationType											X			
choiceList											X			
textDdiContentId												X		
currentStatus													X	
*elem*BitmapentId					X									X

5.12.7 DDI Optional Attributes

The optional DDI attributes (rows) for each DDI element type (columns) are shown in the following table.

Table 11. Optional Attributes of DDI Elements

Optional Attributes	DdiPanel	DdiGroup	DdiPanelLink	DdiButton	DdiBasicButton	DdiToggle	DdiAnimation	DdiShowRange	DdiSetRange	DdiEntry	DdiChoice	DdiText	DdiStatus	DdiIcon
POSITION	X	X	X	X	X	X	X	X	X	X	X	X	X	X
SHOW_WITH_PARENT	X													
INITIAL_FOCUS	X	X												
BACKGROUND_COLOR	X	X				X				X				
BACKGROUND_PICTURE_LINK	X	X												
BACKGROUND_PATTERN	X	X												
FOCUS_SOUND_LINK	X		X	X	X	X	X	X	X	X	X	X		X

FONTSIZE	X	X	X	X	X	X	X	X	X	X	X	X	X
TITLE	X	X											
AUDIO_VIDEO	X	X											
AUDIO	X	X											
SAFETY_AREA_POSITION	X												
FOCUS_NAVIGATION			X	X	X	X	X	X	X	X	X	X		X
HELP_PANEL_LINK			X	X	X	X	X	X	X	X	X	X	X	X
PRESSED_BITMAP_LINK				X										
RELEASED_BITMAP_LINK				X										
PRESSED_SOUND_LINK				X										
RELEASED_SOUND_LINK				X										
SELECT_SOUND_LINK			X		X		X	X				X		X
VALUE_OFFSET								X	X					
VALUE_POWER10								X	X					
MAX_LABEL								X	X					
MIN_LABEL								X	X					
CENTER_LABEL								X	X					
UNIT_LABEL								X	X					
HOTLINK												X		

Note that DEVICE_ICON_BITMAP, CONTENT_ICON_BITMAP, PLAYBACK_DURATION, RECORDED_DATETIME, and BROADCAST_DATETIME are not listed in this table since they are used only by the DeviceIcon and ContentIcon data structures defined in section 5.6.2 *DCM Data Structures*. These optional attributes are not used by any of the DDI elements defined in this section.

OptAttrType

Definition

```
typedef ushort OptAttrType;
```

Description

The particular ushort values for the optional attribute types are specified in Annex 11.16.

OptionalAttribute

```
union OptionalAttribute switch (OptAttrType) {
   case POSITION:              Position      position;
   case SAFETY_AREA_POSITION:  SafetyAreaPosition
                                    safetyAreaPosition;
   case BACKGROUND_COLOR:      DdiColor      backgroundColor;
   case BACKGROUND_PATTERN:    Pattern       backgroundPattern;
   case BACKGROUND_PICTURE_LINK:
                               DdiContentId  backgroundPicture;
   case AUDIO_VIDEO:           AudioVideo    audioVideo;
   case AUDIO:                 Audio         audio;
   case DEVICE_ICON_BITMAP:    Bitmap        deviceIconBitmap;
   case CONTENT_ICON_BITMAP:   Bitmap        contentIconBitmap;
   case PRESSED_BITMAP_LINK:   DdiContentId  pressedBitmap;
   case RELEASED_BITMAP_LINK:  DdiContentId  releasedBitmap;
   case HOTLINK:               wstring<256>  hotlink;
   case FONTSIZE:              Fontsize      fontSize;
```

```
   case FOCUS_NAVIGATION:     FocusNavigation focusNavigation;
   case INITIAL_FOCUS:        DdiElementId  initialfocus;
   case SHOW_WITH_PARENT:     boolean       showWithParent;
   case TITLE:                DdiTitle      titleDdiContentId;
   case VALUE_OFFSET:         short         valueOffset;
   case VALUE_POWER10:        short         valuePower10;
   case MAX_LABEL:            Label         maxLabel;
   case MIN_LABEL:            Label         minLabel;
   case CENTER_LABEL:         Label         centerLabel;
   case UNIT_LABEL:           Label         unitLabel;
   case FOCUS_SOUND_LINK:     DdiContentId  focusSound;
   case PRESSED_SOUND_LINK:   DdiContentId  pressedSound;
   case RELEASED_SOUND_LINK:  DdiContentId  releasedSound;
   case SELECT_SOUND_LINK:    DdiContentId  selectSound;
   case HELP_PANEL_LINK:      DdiElementId  helpPanelLink;
   case PLAYBACK_DURATION:    DateTime      playbackDuration;
   case RECORDED_DATETIME:    DateTime      recordedDateTime;
   case BROADCAST_DATETIME:   DateTime      broadcastDateTime;
};
```

Description

All optional attributes have an associated member of the `OptAttrType` enumerated type.

Those optional attributes that can appear in more than one type of element are described here; some details may also appear in the description of the elements that use them. Optional attributes that can appear in only one type of element are described in the sub-section on that particular element.

OptAttrList

```
typedef sequence<OptionalAttribute> OptAttrList;
```

Description

A list of optional attributes may be empty.

5.12.8 Individual DDI Elements

DdiElement

Definition

```
union DdiElement switch (DdiElementType) {
     case DDI_PANEL:       DdiPanel        panel;
     case DDI_HELP_PANEL:  DdiPanel        helpPanel;
     case DDI_ALERT_PANEL: DdiPanel        alertPanel;
     case DDI_GROUP:       DdiGroup        group;
     case DDI_PANELLINK:   DdiPanelLink    panelLink;
     case DDI_BUTTON:      DdiButton       button;
     case DDI_BASICBUTTON: DdiBasicButton  basicButton;
     case DDI_TOGGLE:      DdiToggle       toggle;
     case DDI_ANIMATION:   DdiAnimation    animation;
     case DDI_SHOWRANGE:   DdiShowRange    showRange;
     case DDI_SETRANGE:    DdiSetRange     setRange;
     case DDI_ENTRY:       DdiEntry        entry;
```

```
      case DDI_CHOICE:      DdiChoice       choice;
      case DDI_TEXT:        DdiText         text;
      case DDI_STATUS:      DdiStatus       status;
      case DDI_ICON:        DdiIcon         icon;
};
```

Description

The DDI element types are specified in Annex 11.15 HAVi DDI Element Types.

DdiPanel

Definition

```
enum AspectRatio {

   // unknown, or non-standard format (pixel aspect ratio)
   UNKNOWN_PIXEL_ASPECT_FORMAT,

   // square pixels (1.0)
   SQUARE_PIXEL_ASPECT_FORMAT,

   // 720 by 576 pixels rendered on a physical
   // 4 by 3 display (1.067)
   PAL_720_BY_576_DISPLAY_4_BY_3_PIXEL_ASPECT_FORMAT,

   // 704 by 480 pixels rendered on a physical
   // 4 by 3 display (0.909)
   NTSC_704_BY_480_DISPLAY_4_BY_3_PIXEL_ASPECT_FORMAT,

   // 720 by 480 pixels rendered on a physical
   // 4 by 3 display (0.889)
   ARIB_720_BY_480_DISPLAY_4_BY_3_PIXEL_ASPECT_FORMAT,

   // 720 by 576 pixels rendered on a physical
   // 16 by 9 display (1.422)
   PAL_720_BY_576_DISPLAY_16_BY_9_PIXEL_ASPECT_FORMAT,

   // 704 by 480 pixels rendered on a physical
   // 16 by 9 display (1.212)
   ATSC_704_BY_480_DISPLAY_16_BY_9_PIXEL_ASPECT_FORMAT,

   // 720 by 480 pixels rendered on a physical
   // 16 by 9 display (1.185)
   ARIB_720_BY_480_DISPLAY_16_BY_9_PIXEL_ASPECT_FORMAT

};


struct DdiPanel {
      DdiElementId       elemId;
      Label              panelName;
      ushort             height;
      ushort             width;
      AspectRatio        aspectRatio;
      DdiElementIdList   elements;
```

```
      OptAttrList         optionals;
};
```

Description

The DdiPanel element can contain groups, panel links, and standalone elements. It cannot contain other panel elements, though. It has it's own DdiElementId. When using the Subscribe operation, the device will return the root panel DdiElementId. All panels are linked from this root panel.

The actual panel size is not specified, but a safety area on which any full capability controller can surely display a panel is 640 x 480 pixels, i.e. the upper-left corner as the user faces the device is <0,0>; the lower-right is <639, 479>. If the panel has the attribute of background AudioVideo, pattern, picture or color, then the background of the panel is displayed on the actual full screen (e.g., 720 x 480 or 720 x 576 etc).

The aspectRatio attribute shows the aspect ratio of the panel and the controller may or may not display the panel using this aspect ratio, i.e. a high grade controller might support all aspect ratios but a low grade controller might support only one aspect ratio for panels.

Optional Attributes

- POSITION – indicates the panel position within the safety area.
- SHOW_WITH_PARENT – The optional attribute showWithParent is used to indicate that this panel, if possible, should be displayed at the same time as its parent panel. The parent panel is defined to be the previous current panel.
- INITIAL_FOCUS – This points to the first DDI Element in the panel to be used for focus navigation.
- BACKGROUND_COLOR
- BACKGROUND_PICTURE_LINK
- BACKGROUND_PATTERN
- FOCUS_SOUND_LINK
- FONTSIZE
- TITLE
- AUDIO_VIDEO
- AUDIO
- SAFETY_AREA_POSITION

Help Panels and Alert Panels

Description

A help panel can be used for explanations of functionality or user operations, etc. that do not fit well on, for example, the current panel. A help panel has the same structure as DdiPanel but its DdiElementType value is DDI_HELP_PANEL.

An alert panel can be used for (large) alerts, dialog boxes, etc. that do not fit well on, for example, the current panel. An alert panel has the same structure as DdiPanel but its DdiElementType value is DDI_ALERT_PANEL.

In the case of help panels or alert panels, the current panel is not changed even if the controller pulls new elements using any of the DDI operations.

DdiGroup

Definition

```
struct DdiGroup {
      DdiElementId       elemId;
      Label              groupName;
      ushort             height;
      ushort             width;
      DdiElementIdList   elements;
      OptAttrList        optionals;
};
```

Description

The `DdiGroup` element is used to suggest to the controller that the elements it contains should be displayed as visually "together". This grouping is helpful when the controller has to scale down the panel. Groups indicate to the controller which elements need to appear together – not necessarily at the same time. A `DdiGroup` has its own `DdiElementId`. Note that panels do not necessarily contain groups.

A `DdiGroup` cannot contain `DdiPanel` elements, but a `DdiGroup` can contain another `DdiGroup` by referencing its `DdiElementId`. However, it is not allowed for targets to construct cyclic references of groups.

Optional Attributes

- `POSITION` – indicates the group position within a panel.
- `INITIAL_FOCUS` – This points to the first DDI Element in the group to be used for focus navigation.
- `BACKGROUND_COLOR`
- `BACKGROUND_PICTURE_LINK`
- `BACKGROUND_PATTERN`
- `FONTSIZE`
- `TITLE`
- `AUDIO_VIDEO`
- `AUDIO`

The background color, picture, audioVideo, and pattern can be chosen to be the background of this group. Only one of these should be present.

DdiPanelLink

Definition

```
struct DdiPanelLink {
      DdiElementId  elemId;
      Label         linkName;
      ushort        height;
      ushort        width;
      Interactivity interactivity;
      DdiContentId  linkBitmap;
      DdiElementId  panelLink;
      OptAttrList   optionals;
};
```

Description

Non-interactive and interactive DdiPanelLinks are intended to indicate to the controller which panel to fetch next. Note that selection of an (interactive or non-interactive) DdiPanelLink does not imply any retrieval of a panel and neither changes the current panel.

The bitmap provided with a DdiPanelLink can be used by the controller to represent this item; how it is presented (as an icon, button, etc.) is up to the controller. If the bitmap is empty, it means that the bitmap is absent.

A suggestion for GUI designers is to include the DdiPanelLink elements together in one group so that the controller can display them together. When displaying a scaled-down GUI, the controller must enable the user to navigate between all DDI elements. How the controller does this is its decision.

The recommended size for the linkBitmap is to allow it to be easily displayed on a screen along with a full size panel. This allows a standard way of displaying DDI elements that are presented specifically to the user for navigation purposes.

Optional Attributes

- POSITION
- FOCUS_SOUND_LINK
- FONTSIZE
- FOCUS_NAVIGATION
- HELP_PANEL_LINK

DdiButton

Definition

```
struct DdiButton {
      DdiElementId  elemId;
      Label         pressedLabel;
      Label         releasedLabel;
      ushort        height;
      ushort        width;
      Interactivity interactivity;
      OptAttrList   optionals;
};
```

Description

The display device determines how the button is presented on the screen. The button element allows two user actions to be placed on it and the controller can determine how this is presented to the user interacting with this DDI element. A press/release button generates a user action when pressed and one when released. A DdiButton is initially in released state, during pushing it is “temporarily” in pressed state. In between the user actions PRESS and RELEASE, no other user action is allowed.

Optional Attributes

- POSITION
- FOCUS_SOUND_LINK
- FONTSIZE
- FOCUS_NAVIGATION
- HELP_PANEL_LINK
- PRESSED_BITMAP_LINK

- RELEASED_BITMAP_LINK
- PRESSED_SOUND_LINK
- RELEASED_SOUND_LINK

DdiBasicButton

Definition

```
struct DdiBasicButton {
      DdiElementId  elemId;
      Label         buttonName;
      ushort        height;
      ushort        width;
      Interactivity interactivity;
      DdiContentId  buttonBitmap;
      OptAttrList   optionals;
};
```

Description

This element is used to display a label or bitmap image for either selection or display only. If the controller displays the bitmap; it is suggested that it does not display the label. An empty bitmap means that the bitmap is absent.

Optional Attributes

- POSITION
- FOCUS_SOUND_LINK
- FONTSIZE
- FOCUS_NAVIGATION
- HELP_PANEL_LINK
- SELECT_SOUND_LINK

DdiToggle

Definition

```
struct ToggleState {
      Label         toggleStateName;
      DdiContentId  toggleStateBitmap;
};

struct DdiToggle {
      DdiElementId          elemId;
      Label                 toggleName;
      ushort                height;
      ushort                width;
      Interactivity            interactivity;
      ushort                numToggleStates;// ( >1 )
      sequence<ToggleState> toggleStates;
      ushort        currentToggleState;// >= 0 and <
      numToggleStates
      OptAttrList   optionals;
};
```

Description

The DdiToggle element allows a choice from several toggle states. A controller is only required to show the current toggle state. The toggleStateName field may be the empty Label. The bitmap referred to by the iconBitmap field may also be empty. This means that the bitmap is absent.

Optional Attributes

- POSITION
- BACKGROUND_COLOR – indicates the color used for the toggle indicator.
- FOCUS_SOUND_LINK
- FONTSIZE
- FOCUS_NAVIGATION
- HELP_PANEL_LINK

DdiAnimation

Definition

```
enum RepetitionType {
   PLAY_ONCE,
   PLAY_REPEATEDLY,
   PLAY_ALTERNATING
};

struct AnimationState {
   Label         animationStateName;
   DdiContentId  animationStateBitmap;
};

struct DdiAnimation {
      DdiElementId              elemId;
      Label                     animationName;
      ushort                    height;
      ushort                    width;
      Interactivity             interactivity;
      RepetitionType            repetition;
      ushort                    animationStateDuration;
      sequence<AnimationState>  animation;
      OptAttrList               optionals;
};
```

Description

This element is a multiple image icon which may be used for presenting an animation.

With PLAY_ALTERNATING the animation starts playing forwards (in the ascending order of the animation sequence) until it reaches the end of the sequence; it then plays the animation sequence backwards until it reaches the beginning and then continues going forwards again.

The controller does not have to display the animationStateNames when displaying the associated bitmaps (if these are present; that is: non-empty).

The animationStateDuration field is the length of time (in units of 0.1 seconds) that each element of the animation sequence should be displayed. The animation sequence should have at least one member.

Optional Attributes

- POSITION
- FOCUS_SOUND_LINK – used only if interactivity = ENABLED.
- FONTSIZE
- FOCUS_NAVIGATION – used only if interactivity = ENABLED.
- HELP_PANEL_LINK
- SELECT_SOUND_LINK

DdiShowRange

Definition

```
enum OrientationType {
   LINEAR_HORIZONTAL,
   LINEAR_VERTICAL,
   CIRCULAR
};

struct DdiShowRange {
      DdiElementId     elemId;
      Label            rangeName;
      ushort           height;
      ushort           width;
      Interactivity    interactivity;
      OrientationType  orientation;
      ushort           valueRange;
      ushort           stepValue;
      ushort           valueSet;
      OptAttrList      optionals;
};
```

Description

It is allowed for controllers to ignore the orientation attribute, however support for at least LINEAR_HORIZONTAL and LINEAR_VERTICAL is highly recommended.

Optional Attributes

- POSITION
- FOCUS_SOUND_LINK – used only if interactivity = ENABLED.
- FONTSIZE
- FOCUS_NAVIGATION – used only if interactivity = ENABLED.
- HELP_PANEL_LINK
- SELECT_SOUND_LINK
- VALUE_OFFSET
- VALUE_POWER10
- MAX_LABEL
- MIN_LABEL
- CENTER_LABEL
- UNIT_LABEL

The values of the valueOffset (= O) and valuePower10 (= P) fields allow a controller to calculate the “real” value by means of the formula: $(V + O)\times 10^{P}$ where V=valueSet.

DdiSetRange

Definition

```
struct DdiSetRange {
      DdiElementId     elemId;
      Label            rangeName;
      ushort           height;
      ushort           width;
      Interactivity    interactivity;
      OrientationType  orientation;
      ushort           valueRange;
      ushort           stepValue;
      ushort           valueSet;
      InformTarget     informTarget;
      OptAttrList      optionals;
};
```

Description

For `OrientationType` see `DdiShowRange.`

Optional Attributes

- `POSITION`
- `FOCUS_SOUND_LINK` – used only if `interactivity` = `ENABLED`.
- `FONTSIZE`
- `FOCUS_NAVIGATION` – used only if `interactivity` = `ENABLED`.
- `HELP_PANEL_LINK`
- `VALUE_OFFSET`
- `VALUE_POWER10`
- `MAX_LABEL`
- `MIN_LABEL`
- `CENTER_LABEL`
- `UNIT_LABEL`

The values of the `valueOffset` (= O) and `valuePower10` (= P) fields allow a controller to calculate the "real" value by means of the formula: $(V + O)\times 10^{P}$ where V= the current value in range.

DdiEntry

Definition

```
enum EntryType { TEXTUAL, NAT_NUMBER, FLOAT, DATE, TIME};

union EntryFormats switch ( EntryType ) {
   case TEXTUAL:    wstring<256>  entryString;
   case NAT_NUMBER: uint          entryLiteral;
   case FLOAT:      float         entryFloat;
   case DATE:       DateTime      entryDate;
   case TIME:       DateTime      entryTime;
};

enum Qualifier {CONCEAL, VISIBLE};
struct DdiEntry {
      DdiElementId  elemId;
```

```
        Label         entryName;
        ushort        height;
        ushort        width;
        Interactivity interactivity;
        EntryType     entryType;
        Qualifier     qualifier;
        InformTarget  informTarget;
        EntryFormats  currentEntry;
        ushort        maxCharsDigits;
        OptAttrList   optionals;
};
```

Description

In the case of `entryType DATE`, only the [year/month/day] fields of the `DateTime` structure can be used; the other `DateTime` fields are ignored.

In the case of `entryType TIME`, only the [hour/min/sec] fields of the `DateTime` structure can be used; the other `DateTime` fields are ignored.

The `Qualifier` attribute determines whether characters within the entry field should be displayed or concealed, e.g. for PIN entry.

The `maxCharsDigits` field contains the maximum number of characters or digits that can be entered by the user. The field should be ignored for the `FLOAT`, `DATE` and `TIME` entry types.

Optional Attributes

- `POSITION`
- `BACKGROUND_COLOR` – indicates the color used for the entry indicator.
- `FOCUS_SOUND_LINK` – used only if `interactivity` = `ENABLED`.
- `FONTSIZE`
- `FOCUS_NAVIGATION` – used only if `interactivity` = `ENABLED`.
- `HELP_PANEL_LINK`

DdiChoice

Definition

```
struct ChoiceElement {
   Label         choiceName;
   DdiContentId  choiceBitmapDdiContentId;
   boolean       elementStatus;
};

enum ChoiceType {LESS_THAN, EQUAL, MORE_THAN};

enum WrapType {DONT_CARE, STOP_AT_BORDERS, WRAP_AROUND};

enum ChoiceOrientationType {
   HORIZONTAL,
   VERTICAL
};


struct DdiChoice {
```

```
      DdiElementId              elemId;
      Label                     choiceName;
      ushort                    height;
      ushort                    width;
      Interactivity             interactivity;
      ChoiceType                choiceType;
      ushort                    choiceNumber;
      WrapType                  wrapType;
      ChoiceOrientationType     choiceOrientationType;
      sequence<ChoiceElement>   choiceList;
      OptAttrList               optionals;
};
```

Description

Refer to the user action associated with the choice DDI element to see how it returns the list of choices made or only returns the single choice (radio button). The `elementStatus` field of the `ChoiceElement` structure indicates the current status of this choice element, i.e. `True` indicates chosen, `False` indicates not chosen. A graphics capable controller does not need to show the labels when displaying the supplied bitmaps (if these are present; that is: non-empty).

A controller that cannot show all choice elements at once, shows only a subset. Depending on the `wrapType` attribute. For the value `WRAP_AROUND`, the controller continues with showing the first elements next to the last elements (and the other way around). For the value `STOP_AT_BORDERS`, the controller stops showing at border (first and last) elements. The `choiceOrientationType` attribute indicates whether the choice elements should be rendered next to each other or below each other. A controller is allowed to ignore this attribute, however support for this attribute is highly recommended.

The `choiceNumber` together with the `choiceType` define how many elements from the total number of elements the user is expected to select.

The `choiceList` sequence should not be empty.

Optional Attributes

- `POSITION`
- `FOCUS_SOUND_LINK` – used only if `interactivity` = `ENABLED`.
- `FONTSIZE`
- `FOCUS_NAVIGATION` – used only if `interactivity` = `ENABLED`.
- `HELP_PANEL_LINK`

DdiText

Definition

```
struct DdiText {
      DdiElementId  elemId;
      ushort        height;
      ushort        width;
      Interactivity interactivity;
      DdiContentId  textContent;
      OptAttrList   optionals;
};
```

Description

The text DDI element is used if the device only wants to display text, with no user entry of text. Textual data can contain newline characters and can have at most 40 characters horizontally; a vertical (line) limit is not needed since the controller can use scrolling if necessary. `DdiText` can also be used as an interactive field that would be a text string that is a "hot link".

Optional Attributes

- `POSITION`
- `FOCUS_SOUND_LINK` – used only if `interactivity` = `ENABLED`.
- `FONTSIZE`
- `FOCUS_NAVIGATION` – used only if `interactivity` = `ENABLED`.
- `HELP_PANEL_LINK`
- `SELECT_SOUND_LINK`
- `HOTLINK` – if present indicates that the text contains a URL address. It is a controller option whether to link to the addressed site or not.

DdiStatus

Definition

```
struct DdiStatus {
      DdiElementId  elemId;
      Label         statusName;
      ushort        height;
      ushort        width;
      ushort        currentStatus;
      OptAttrList   optionals;
};
```

Description

This element is used to inform the controller and user that the target (device) is "busy". This would provide GUI functions such as "hourglass", "barber pole", incrementing bar, etc.

The `currentStatus` field indicates if the controller should show the element as "working" (=1) or not working (=0). A controller is allowed to obscure (that is: not render) a "not working" Status element. It is still useful for targets to specify such kind of element for reserving already the required space when it is eventually switched to "working".

Optional Attributes

- `POSITION`
- `FONTSIZE`
- `HELP_PANEL_LINK`

DdiIcon

Definition

```
struct DdiIcon {
      DdiElementId  elemId;
      Label         iconName;
      ushort        height;
      ushort        width;
      Interactivity interactivity;
      DdiContentId  iconBitmap;
      OptAttrList   optionals;
```

```
};
```

Description

This element is used to display a bitmap image for either selection or display only. If the controller displays the bitmap; it is suggested that it does not display the label. An empty bitmap means that the bitmap is absent.

Optional Attributes

- POSITION
- FOCUS_SOUND_LINK – used only if interactivity = ENABLED.
- FONTSIZE
- FOCUS_NAVIGATION – used only if interactivity = ENABLED.
- HELP_PANEL_LINK
- SELECT_SOUND_LINK

5.12.9 DDI Action Data Structures

DDI actions are only defined for interactive DDI elements. Actions can be placed on DDI elements when the DDI element has the controller focus. How and when the focus is placed on a particular DDI element is up to the controller. Typically, the focus would be shown by highlighting a box around the DDI element or changing its color.

The DDI actions correspond to the DDI element types. In this way, the UserAction operation can check the action taken is on the correct type of element.

ActType

Definition

```
enum ActType {
   ACT_BUTTON,   ACT_TOGGLE,   ACT_SETRANGE,    ACT_ENTRY,
   ACT_CHOICE,   ACT_SELECTED, ACT_ANIMATION
};
```

Description

These action types correspond with the DDI element types in the following way:

Table 12. DDI Action Types

Action Type (ActType)	DdiPanel	DdiGroup	DdiPanelLink	DdiButton	DdiBasicButton	DdiToggle	DdiAnimation	DdiShowRange	DdiSetRange	DdiEntry	DdiChoice	DdiText	DdiStatus	DdiIcon
ACT_BUTTON				X										
ACT_TOGGLE						X								
ACT_ANIMATION							X							
ACT_SELECTED			X		X			X				X		X
ACT_SETRANGE									X					
ACT_ENTRY										X				
ACT_CHOICE											X			

Note that the user actions of the indicated types only hold for DDI elements with the `interactivity` attribute set to `ENABLED`.

ActButton

```
enum ActButton {PRESS, RELEASE};
```

ActToggle

```
typedef ushort ActToggle;
```

Indicates the state that has been set (>= 0 and < `nr_of_states`).

ActAnimation

```
typedef short ActAnimation;
```

Description

This value indicates the sequence number (starting from 0) of the `AnimationElement` that the user has selected. In case the `RepetitionType` is `PLAY_ONCE`, the value -1 indicates that the complete animation sequence has been shown to the user without selecting any specific element (note that this is not a genuine user action). Any further (genuine) user selections will result in an indication of the last `AnimationElement`.

ActSetRange

```
typedef ushort ActSetRange;
```

Indicates the value that has been set (>= 0 and <= `value_range`).

ActEntry

```
typedef EntryFormats ActEntry;
```

Indicates the entry value.

ActChoiceList

```
typedef sequence<ushort> ActChoiceList;
```

Each number in `ActChoiceList` corresponds with the index of the “checked” entry; in case of single choice, `ActChoiceList` contains only one number (the one selected).

ActSelected

```
typedef boolean ActSelected;  // dummy value
```

DdiAction

```
union DdiAction switch (ActType) {
   case ACT_BUTTON:      ActButton button;
   case ACT_TOGGLE:      ActToggle toggle;
   case ACT_ANIMATION:   ActAnimation animation;
   case ACT_SETRANGE:    ActSetRange setRange;
   case ACT_ENTRY:       ActEntry entry;
```

```
   case ACT_CHOICE:      ActChoiceList choiceList;
   case ACT_SELECTED:    ActSelected selected;
};
```

5.12.10 Resource Limitations

The following safe parameter size limits apply to DDI elements and associated data structures:

Table 13. DDI Resource Limitations

DDI Data Structure	Safe Parameter Size Limit
`DdiElementIdList`	defined by the relevant elements and APIs
`DdiElementList`	defined by the relevant elements and APIs
`Bitmap`	no limitation imposed by DDI
`Sound`	no limitation imposed by DDI
`Label`	`wstring<16>`
`UnicodeText`	`wstring<1024>`
`Hotlink`	`wstring<256>`
`ActEntry`	`wstring<256>`
`ActChoiceList`	64 `ushort` values
`OpAttrList`	16 `OptionalAttribute` values
`DdiPanel.elements`	256 `DdiElementId` values
`DdiGroup.elements`	256 `DdiElementId` values
`DdiAnimation`	32 `AnimationElement` values
`DdiChoice.choiceList`	64 `ChoiceElement` values

5.12.11 Data Driven Interaction API

DdiTarget::Subscribe

Prototype

```
Status DdiTarget::Subscribe(
      in    OperationCode       opCode,
      in    NotificationScope   scope,
      out   DdiElementId        rootPanel
)
```

Parameters

- `OpCode` – the `OperationCode` provided by the controller. This is the value that the target will place in the operation code of a `NotifyDdiChange` message it sends to the controller.
- `scope` – indicates whether the target should generate DDI change reports only for the current panel and the elements within it (=`CURRENT`), for all panels and the elements within them (=`GLOBAL`) or for elements within the set of panels retrieved since the last call to `DdiTarget::Subscribe` or `DdiTarget::ChangeScope` (= `ADD`). The current panel is always part of the notification scope.
- `rootPanel` – the `DdiElementId` of the initial (root) panel.

Description

Indicates to the target that this controller is starting a DDI subscription; i.e., that the target's DDI is going to be used by this controller. The target returns an initial or *root panel*. Note, the root panel

need not be the same for every `Subscribe`. A `Subscribe` also indicates to the target to which controller it has to send back `NotifyDdiChange` messages. Note, the controller can use the fact that the SEID of the sender of a message – here the controller – is always part of a message and is available to the receiver – here the target (see the `MsgCallback` callback in section 5.3.3 *Messaging System API*). In the case where the target is a DCM, `Subscribe` also enables that DCM to tell other system components whether it is "in use" (e.g., to the DCM Manager which might want to uninstall the DCM).

After subscription the "current panel" is defined to be the root panel. Thereafter the "current panel" is the panel which the controller most recently pulled using `GetDdiPanel`, `GetDdiElement` or `GetDdiElementList`. If more than one panel is pulled by the `GetDdiElementList` then the panel that is the last in the list will be the current panel.

If the controller tries to pull a DDI element not within the current notification scope (`CURRENT or ADD`), the element is returned but the error code `DdiTarget::ENOT_CUR` is returned warning that the controller will not receive change notifications for this DDI element.

Error codes

- `DdiTarget::ENO_SUB` – no DDI subscription is possible; e.g., the target has run out of the resources necessary to keep track of another simultaneous subscription

DdiTarget::Unsubscribe

Prototype

```
Status DdiTarget::Unsubscribe()
```

Description

Indicates to the target that a currently open subscription from this controller has ended. The target will no longer send `NotifyDdiChange` messages to this controller.

Error codes

- `DdiTarget::ENO_SUB` – no subscription for the DDI of this target is currently open from this controller

DdiTarget::GetDdiElement

Prototype

```
Status DdiTarget::GetDdiElement(
      in    DdiElementId  elementId,
      out   DdiElement    element
)
```

Parameters

- `elementId` – the `DdiElementId` of the requested DDI element.
- `element` – the requested DDI element returned.

Description

Pulls the requested DDI element from the target.

Error codes

- `DdiTarget::ENO_DEI` – unknown `DdiElementId`

- `DdiTarget::ENO_SUB` – no subscription for the DDI of this target is currently open from this controller
- `DdiTarget::ENOT_CUR` – scope violation, the pulled DDI element is not within the current notification scope; the element, however, is still returned

DdiTarget::GetDdiPanel

Prototype

```
Status DdiTarget::GetDdiPanel(
      in    DdiElementId     elementId,
      out   DdiPanel         panel,
      out   DdiElementList   elementList
)
```

Parameters

- `elementId` – the `DdiElementId` of a panel element.
- `panel` – the panel referred to by `elementId`.
- `elementList` – the elements contained in that panel list for the DDI element structures. The order of the elements in `elementList` must correspond to the order of `elements` in the `DdiPanel` structure specified by `elementId`. The safe parameter size limit is 256 `DdiElement` values.

Description

Pulls the requested panel and its DDI elements from the target. Note that only elements directly associated with the panel are pulled (i.e., elements associated with any groups inside the panel are not pulled).

Error codes

- `DdiTarget::ENO_DEI` – unknown `DdiElementId`
- `DdiTarget::ENO_PANEL` – `elementId` is not the `DdiElementId` of a panel element
- `DdiTarget::ENO_SUB` – no subscription for the DDI of this target is currently open from this controller

DdiTarget::GetDdiGroup

Prototype

```
Status DdiTarget::GetDdiGroup(
      in    DdiElementId     elementId,
      out   DdiGroup         group,
      out   DdiElementList   elementList
)
```

Parameters

- `elementId` – the `DdiElementId` of a group element.
- `group` – the group referred to by `elementId`.
- `elementList` – the elements contained in that group list for the DDI element structures. The order of the elements in `elementList` must correspond to the order of `elements` in the `DdiGroup` structure specified by `elementId`. The safe parameter size limit is 256 `DdiElement` values.

Description

Pulls the requested group and its DDI elements from the target. Note that only elements directly associated with the group are pulled (i.e., elements associated with any groups inside the group are not pulled).

Error codes

- `DdiTarget::ENO_DEI` – unknown `DdiElementId`
- `DdiTarget::ENO_GROUP` – `elementId` is not the `DdiElementId` of a group element
- `DdiTarget::ENO_SUB` – no subscription for the DDI of this target is currently open from this controller
- `DdiTarget::ENOT_CUR` – notification scope violation, the pulled group element is not within the current panel; the group and its elements, however, are still returned

DdiTarget::GetDdiElementList

Prototype

```
Status DdiTarget::GetDdiElementList(
      in    DdiElementIdList elementIdList,
      out   DdiElementList   elementList
)
```

Parameters

- `elementIdList` – an arbitrary list of `DdiElementId` values. The safe parameter size limit is 256 `DdiElementId` values.
- `elementList` – the corresponding list of DDI element structures returned. The safe parameter size limit is 256 `DdiElement` values.

Description

Pulls the requested arbitrary list of DDI elements from the device. Any type of DDI element can be retrieved with this operation, including `DdiPanel` and `DdiGroup`. It is up to the controller how to use the data once retrieved. In the case of `ADD` notification scope mode, all loaded panels should be added to the scope of the DDI Controller.

If one or more of the `DdiElementId` are unknown from the target, the `ENO_DEI` error code is returned. In such case, the output element list contains all the elements that could be found

Error codes

- `DdiTarget::ENO_DEI` – unknown `DdiElementId`
- `DdiTarget::ENO_SUB` – no subscription for the DDI of this target is currently open from this controller
- `DdiTarget::ENOT_CUR` – notification scope violation, one or more of the pulled DDI elements are not within the current panel; the elements, however, are still returned

DdiTarget::GetDdiContent

Prototype

```
Status DdiTarget::GetDdiContent (
      in    DdiContentId  ceid,
      out   DdiContent    content
)
```

Parameters

- `ceid` – the `DdiContentId` of the requested content.
- `content` – the requested content (i.e. text, bitmap or sound data) returned.

Description

Pulls the requested content from the target.

Error codes

- `DdiTarget::ENO_DEI` – unknown `DdiContentId`
- `DdiTarget::ENO_SUB` – no subscription for the DDI of this target is currently open from this controller

DdiTarget::ChangeScope

Prototype

```
Status DdiTarget::ChangeScope (
      in    NotificationScope scope
)
```

Parameters

- `scope` – the new scope to be used by the DDI Target for this DDI Controller

Description

Modifies on the target side the notification scope of the controller. The initial scope after the call of `DdiTarget::ChangeScope` contains only the current panel except when set to `GLOBAL` which contains all panels.

Error codes

- `EINVALID_PARAMETER` – invalid `scope`
- `DdiTarget::ENO_SUB` – no subscription for the DDI of this target is currently open from this controller

DdiTarget::UserAction

Prototype

```
Status DdiTarget::UserAction(
      in    DdiElementId       elementId,
      in    DdiAction          action,
      out   DdiElementId       targetPanel,
      out   DdiElementIdList   report,
      out   DdiElementIdList   deletedPanelList
)
```

Parameters

- `elementId` – the `DdiElementId` of the DDI element associated with the user action.
- `action` – the user action information.
- `targetPanel` – the panel that the target suggests to the controller. It is recommended that the controller retrieve the `targetPanel` (if not already retrieved beforehand) and show it to the user.
- `report` – the change report for this action: the list of `DdiElementId`'s of DDI elements that changed due to this user action. The safe parameter size limit is 256 `DdiElementId` values.
- `deletedPanelList` – list of `DdiElementId`'s of type `DdiPanel` that are no longer

available at the target.

Description

Sent by the controller to the target to indicate the user action the controller performed on the specified DDI element. The response from the target indicates which DDI elements changed due to this user action. Implicit changes in DDI elements that are already known to the controller (e.g., the value change in a DdiSetRange element) should not be reported by the target to that controller, if the target accepts the action. If the target does not accept, the report should indicate the DDI element with the correct value. The target however needs to take these changes into account (if the user action was accepted).

The `report` result will have a value based on the notification scope requested by the controller. In the case where `scope` equals `CURRENT`, `report` will only contain changes within the current panel. In the case where `scope` equals `GLOBAL`, `report` will contain all changes within the target DDI data. In the case where `scope` equals `ADD`, `report` will contain changes within the set of panels retrieved since the last call to `DdiTarget::Subscribe` or `DdiTarget::ChangeScope`. In case of no changes, `report` will be empty.

Any non-organizational target DDI element that changes and is within the current notification scope will be included in the change report. If DDI elements are added to or removed from organizational target DDI elements then these organizational elements will be included in the change report sent by the target to the controller. If both non-organizational and organizational target DDI elements are changed, both changes will be included.

A `targetPanel` can be used by the target for (large) alerts, dialog boxes, etc. that do not fit well on, for example, the current panel. In case the controller already retrieved this panel beforehand (e.g. in case `scope` equals `GLOBAL`) it still indicates that this panel contains important information for the user. The `targetPanel` can, but does not need to be, a different panel than the current panel (the panel most recently retrieved by the controller).

GUI updates can be reported by the target to the controller via `UserAction` and/or `NotifyDdiChange`. However, the DDI Controller shall not send the next message to the DDI Target until it receives the report of the `UserAction`.

The `deletedPanelList` contains a list of “deleted” panels. A controller should not render these panels anymore. It is recommended for the controller to render the target panel instead. It is required for targets that no valid panel contains a panel link to a deleted panel. A `deletedPanelList` can of course be empty and should not contain the root panel.

Error codes

- `EINVALID_PARAMETER` – invalid `action`
- `DdiTarget::ENO_DEI` – unknown `DdiElementId`
- `DdiTarget::ENO_SUB` – no subscription for the DDI of this target is currently open from this controller
- `DdiTarget::ENOT_CUR` – notification scope violation, the specified DDI element is not within the current notification scope

<Client>::NotifyDdiChange

Prototype

```
Status <Client>::NotifyDdiChange(
      in DdiElementId       targetPanel,
```

```
        in DdiElementIdList    report,
        in DdiElementIdList    deletedPanelList
)
```

Parameters

- `targetPanel` – the panel that the target suggests to the controller. It is recommended that the controller retrieve the `targetPanel` (if not already retrieved beforehand) and show it to the user.
- `report` – the change report for this notification: the list of `DdiElementId`'s of DDI elements that changed (since the previous change report within the current subscription or, if this is the first change report, since subscription) in the target. The safe parameter size limit is 256 `DdiElementId` values.
- `deletedPanelList` – list of `DdiElementId`'s of type `DdiPanel` that are no longer available at the target.

Description

During the subscription, the target (`Client`) can send a "message back" to the subscribing controller. Such a `NotifyDdiChange` message is sent by the target to the controller to provide a change report that indicates which of the target's DDI elements have changed "spontaneously". Note, the target knows the controller from the sender's SEID contained in the message (`DdiTarget::Subscribe`), that is required to have been sent by the controller to the target in order to open the current subscription with the target. The operation code value to be used for the `NotifyDdiChange` message is the `opCode` parameter of `DdiTarget::Subscribe.`

The `report` result will have a value based on the notification scope requested by the controller. In the case where `scope` equals `CURRENT`, `report` will only contain changes within the current panel. In the case where `scope` equals `GLOBAL`, `report` will contain all changes within the target DDI data. In the case where `scope` equals `ADD`, `report` will contain changes within the set of panels retrieved since the last call to `DdiTarget::Subscribe` or `DdiTarget::ChangeScope`. In case of no changes, a notification with an empty `report` can be used by the target to indicate a (different) `targetPanel`.

Any non-organizational target DDI element that changes and is within the current notification scope will be included in the change report. If DDI elements are added to or removed from organizational target DDI elements then these organizational elements will be included in the change report sent by the target to the controller. If both non-organizational and organizational target DDI elements are changed, both changes will be included.

A `targetPanel` can be used by the target for (large) alerts, dialog boxes, etc. that do not fit well on, for example, the current panel. In case the controller already retrieved this panel beforehand (e.g. in case `scope` equals `GLOBAL`) it still indicates that this panel contains important information for the user. The `targetPanel` can, but does not need to be, a different panel than the current panel last retrieved by the controller.

The `deletedPanelList` contains a list of "deleted" panels. A controller should not render these panels anymore. It is recommended for the controller to render the target panel instead. It is required for targets that no valid panel contains a panel link to a deleted panel. A `deletedPanelList` can of course be empty and should not contain the root panel.

5.13 APIs for Versioning

5.13.1 Services Provided

Service	Comm Type	Locality	Access
Version::GetVersion	M	global	all

5.13.2 Version Control API

Version::GetVersion

Prototype

```
Status Version::GetVersion(out Version version)
```

Description

All software elements are required to support `GetVersion`. This function returns the version number of the software element which is queried. This value will reflect the version of the HAVi specification to which the software element was designed.

`GetVersion` returns the version information of the software element which implements this function. For display purposes versions are written as *`major.minor`* in base 10. These rules shall be followed:

- The major value ranges from 1 to 255 (inclusive) with no leading zeros
- The minor value ranges from 0 to 99 (inclusive)
  - Shall be displayed with a leading zero for values between 1 and 9, inclusive
  - Shall never have any trailing zeros

Example Chart for Version Control

major	*minor*	display
1	0	1.0
1	1	1.01
2	5	2.05
20	12	20.12
20	50	20.5
20	100	illegal

The version value is encoded into four bytes as follows:

Table 14. Version Number Encoding

msb byte 3	byte 2	byte 1	lsb byte 0
0x00	major	0x00	minor

As an example, 00 06 00 0C hex would indicate Version 6.12.

5.14 APIs for Bulk Transfer

The Messaging System does not guaranty the ability to transfer very large messages. “Very large” can have different meanings according to the target (FAV or IAV) design. Since IAVs implement HAVi in full native code and according to the TAM specification, bulk transfer (with restrictions) is possible by providing flow control through the IEEE 1394 specification. However for FAVs it is more difficult since there is no way the Messaging System can provide flow control for a target software element. Consequently the TAM receiving a very large message may not be able to allocate memory as needed.

The problem is solved through an API which takes place over the Messaging System. Any component which participates in bulk data transfers, is recommended to design its API according to the following rules:

- a producer API has to offer a way for the client to indicate the maximum size of incoming messages it can accept
- a consumer API has to offer a way for the client to indicate the maximum size of incoming messages it can accept
- both APIs have to offer a way to partition bulk data transfers into a set of messages

According to the previous rules, the following example shows the design of an API which is able to manage bulk transfer.

This hypothetical API allows any client to send (receive) bulk data to (from) a container. Only one container can be open at a time. Since several containers may exist, the client must first perform an `OpenContainer` call. Through this call the client gives the name of container to open, the maximum message size it can process and an operation code the target module will use to send to the client (see 5.1.1– “Message Back”). The target module returns back the maximum message size it can process.

Once the `OpenContainer` call is performed the client can start bulk data transfer using the `WriteInContainer` call. The client will perform this call as often as necessary – i.e., the bulk data to be written is split into several messages. Each message size has to fit within the maximum size constraint given by the target module as the return value of the `OpenContainer` call. A parameter is used to signal to the target whether the transfer is starting, running or done. To control the incoming flow the target module can send the response (status) of the incoming call (`WriteInContainer`) when it is ready to accept the next part of the bulk data transfer.

The target module can also start bulk data transfer to the client. It calls the method identified by the operation code previously given by the client through the `OpenContainer` call. The target module will perform this call as many times as needed – i.e. the bulk data to be written to the client will be split into several messages. Each message size has to fit within the maximum size constraint given by the client as input parameter of the `OpenContainer` call. A parameter is used to signal to the client whether the transfer is starting, running or done. To control the incoming flow the client can send the response (status) of the incoming call when it is ready to accept the next part of bulk data transfer.

The client can stop a transfer at any time using `CloseContainer`.

```
enum { START, MIDDLE, END } Location;
//
// if the data fits into one message
```

```
// the END value will be used

Status OpenContainer(
      in wstring containerName,
      in long clientMessageMaxSize,
      in OperationCode clientOpCode,
      out long targetMessageMaxSize)
//
// Status return code: SUCCESS, BADNAME, ALREADYOPEN

Status WriteInContainer(
      in Location loc,
      sequence<octet> dataPart)
//
// Status return code: SUCCESS, NOTAVAILABLE

void CloseContainer(in wstring containerName)
```

The following prototype has to be implemented by the client to allow the target module to send bulk data transfers:

```
Status <clientOpCode>(
      in Location loc,
      in sequence<octet> dataPart)
//
// Status return code: SUCCESS, NOTAVAILABLE
```

6 APIs for Functional Component Modules

In this section, APIs for Functional Component Modules will be described. These APIs are, for example, APIs for controls specific to the VCR functions within a device, such as PLAY, RECORD, etc.

In the HAVi Architecture, these APIs are sent from an application to the Functional Component Modules, and translated to a native language there. The native language command is then sent from the FCM to the appropriate target device.

6.1 FCM Data Types

ForwardSpeed

```
enum ForwardSpeed {
      PLAY_PAUSE,
      SLOWEST_FORWARD,
      SLOW_FORWARD_5,
      SLOW_FORWARD_4,
      SLOW_FORWARD_3,
      SLOW_FORWARD_2,
      SLOW_FORWARD_1,
      FAST_FORWARD_1,
      FAST_FORWARD_2,
      FAST_FORWARD_3,
      FAST_FORWARD_4,
      FAST_FORWARD_5,
      FASTEST_FORWARD
};
```

`ForwardSpeed` values represent forward speed modes of tape and disc devices. Let *X1* be normal play speed. The actual speeds encoded by the these values have the following restrictions:

- *X1* ≤ `FAST_FORWARD_1` ≤ `FAST_FORWARD_2` ≤ `FAST_FORWARD_3` ≤ `FAST_FORWARD_4` ≤ `FAST_FORWARD_5` ≤ `FASTEST_FORWARD`

- `SLOWEST_FORWARD` ≤ `SLOW_FORWARD_5` ≤ `SLOW_FORWARD_4` ≤ `SLOW_FORWARD_3` ≤ `SLOW_FORWARD_2` ≤ `SLOW_FORWARD_1` ≤ *X1*

`PLAY_PAUSE` indicates the mode in which the device continuously produces a stream consisting of the same frame of video content. It is usually called “still” mode.

ReverseSpeed

```
enum ReverseSpeed {
      SLOWEST_REVERSE,
      SLOW_REVERSE_5,
      SLOW_REVERSE_4,
      SLOW_REVERSE_3,
      SLOW_REVERSE_2,
      SLOW_REVERSE_1,
```

```
      X1_REVERSE,
      FAST_REVERSE_1,
      FAST_REVERSE_2,
      FAST_REVERSE_3,
      FAST_REVERSE_4,
      FAST_REVERSE_5,
      FASTEST_REVERSE
};
```

`ReverseSpeed` values represent reverse speed modes of tape and disc devices. The actual speeds encoded by the these values have the following restrictions (`X1_REVERSE` indicates normal reverse speed):

- `X1_REVERSE` ≤ `FAST_REVERSE_1` ≤ `FAST_REVERSE_2` ≤ `FAST_REVERSE_3` ≤ `FAST_REVERSE_4` ≤ `FAST_REVERSE_5` ≤ `FASTEST_REVERSE`

- `SLOWEST_REVERSE` ≤ `SLOW_REVERSE_5` ≤ `SLOW_REVERSE_4` ≤ `SLOW_REVERSE_3` ≤ `SLOW_REVERSE_2` ≤ `SLOW_REVERSE_1` ≤ `X1_REVERSE`

SkipDirection

```
enum SkipDirection { FORWARD, REVERSE };
```

`SkipDirection` controls the direction of skipping in tape and disc devices.

SkipMode

```
enum SkipMode {
      FRAME, SCENE, TRACK,
      VISS, GOP, INDEX, SKIP,
      PHOTO_PICTURE, PROGRAM,
      MARKER, RELATIVE_TIME, ABSOLUTE_TIME
};
```

`SkipMode` controls the unit of skipping in tape and disc devices.

- `FRAME` – video frames
- `SCENE` – video scenes
- `TRACK` – tracks (disc devices only)
- `VISS` – VHS Index Search System (tape devices only)
- `GOP` – MPEG “group of pictures”
- `INDEX` – index values (an index indicates the position marker used to search for the starting position of a recorded program, or a user-specified position in a program. In the case of DVCR, an index indicates an index ID defined by the HD Digital VCR Conference. In the case of D-VHS VCR, an index indicates an index flag recorded on the medium. For other devices, `INDEX` and `MARKER` may be equivalent).
- `SKIP` – skip over discarded regions
- `PHOTO_PICTURE` – photo/picture
- `PROGRAM` – program units
- `MARKER` – markers (a marker indicates the position marker used to search for the starting position of a data area specified by the user. In the case of D-VHS VCR, a marker indicates a marker flag recorded on the medium. For other devices, `INDEX` and `MARKER` may be equivalent).
- `ABSOLUTE_TIME` – time (hours:minutes:seconds:frame) from the beginning of the tape or disc
- `RELATIVE_TIME` – time (hours:minutes:seconds:frame) from the beginning of the current

track (disc devices only)

TimeCode

```
struct TimeCode {
      octet    hour;
      octet    minute;
      octet    sec;
      octet    frame;
};
```

Time code of the form HH:MM:SS:FF (hours:minutes:seconds:frames). Each of the HH,MM,SS,FF is encoded in 8-bit HEX format. Thus, the size of a time code value is 32 bits. The most significant bit of `hour` indicates a sign (minus or plus), where the value of 1 indicates a minus sign.

Example:
"0x01172D0C" = (+) 01h:23m:45s:12f
"0x81172D0C" = (-) 01h:23m:45s:12f

Valid values for HH: 0-127
Valid values for MM: 0-59
Valid values for SS: 0-59

The range of FF is:
0 - 24 for PAL-based video content
0 - 29 for NTSC-based video content (drop frame)
0 - 99 for audio-only content

If an invalid `TimeCode` value is specified as an in parameter to an API call, an `EINVALID_PARAM` error code may be returned.

WriteProtectStatus

```
enum WriteProtectStatus {
      WRITABLE,
      WRITE_PROTECTED,
      NOT_WRITEABLE,
      UNKNOWN_WRITABLE
};
```

- `WRITABLE` – OK to record on medium
- `WRITE_PROTECTED` – loaded medium is writable but write protected
- `NOT_WRITEABLE` – loaded medium is not writable (i.e. read-only medium)
- `UNKNOWN_WRITABLE` – writable status cannot be determined

6.2 Tuner FCM

The Tuner FCM APIs are meant to be applicable to a wide variety of tuners – from tuners used for analog radio and television to digital tuners such as those for ATSC, DVB and DSS.

A Tuner FCM is capable of producing one or more *service lists*. Each entry in a service list identifies a selectable entity called a *service*. The act of *selecting* a service results in a stream of some type being associated with an output plug of the FCM. *It should be stressed that "service" is used here in a very general sense and encompasses both multiplexes of broadcast programs and components of broadcast programs.* For example, a service could be a multiplex containing several AV streams and data streams (i.e., several broadcast programs), an AV multiplex (i.e., a single broadcast program), or simply an elementary stream (i.e., a component of a broadcast program). Whether a Tuner FCM supports selection of program multiplexes and/or program components is tuner and Tuner FCM dependent. A Tuner FCM may implement an optional API which exposes the relationship between a multiplex and its components.

Some tuners are capable of providing descriptions of specific programs being broadcast or scheduled to be broadcast. For example, digital television tuners can determine the name and start/stop times of programs, while some audio tuners can determine the title and artist of music they receive. These descriptions are called *service events*. An optional API of the Tuner FCM provides access to a simple form of service event information. (Service events should not be confused with HAVi events; service events are merely descriptive and do not correspond to, or trigger, HAVi events.)

6.2.1 Tuner Services

Service	Comm Type	Locality	Access	Resv Prot
Tuner::GetServiceListInfo	M	global	all	
Tuner::GetServiceList	M	global	all	
Tuner::SetServiceList	M	global	all	yes
Tuner::GetService	M	global	all	
Tuner::GetServiceComponents	M	global	all	
Tuner::GetServiceEvents	M	global	all	
Tuner::SelectService	M	global	all	yes
Tuner::GetSelectedServices	M	global	all	
Tuner::GetCapability	M	global	all	
TunerServiceChanged	E	global	Tuner (all)	

6.2.2 Tuner Data Structures

ServiceListType

```
enum ServiceListType {
        TUNER_ASSOCIATED_NAMES,
        PROVIDER_ASSOCIATED_NAMES,
        USER_ASSOCIATED_NAMES,
};
```

Description

Each entry in a service list has a name. `ServiceListType` indicates the meaning and possible usage of the names in the particular service list. For `TUNER_ASSOCIATED_NAMES`, the name is assigned by the tuner or Tuner FCM and may be directly associated with underlying tuning frequencies, for example “Channel 4” or “88.5 MHz”. For `PROVIDER_ASSOCIATED_NAMES`, the name identifies the service using a well known commercial name or brand, for example “CNN” or “BBC-1”. Often such a name is obtained from the service provider. `TUNER_ASSOCIATED_NAMES` and `PROVIDER_ASSOCIATED_NAMES` give some means for interoperability between different tuners (even between tuners of different type). Finally, `USER_ASSOCIATED_NAMES` are private names chosen by a user of the tuner. A Tuner FCM is not obliged to provide service lists of all types, but it shall provide at least one service list. Tuners may provide several lists of the same type. Within a service list, the names of entries need not be unique unless the list is of type `TUNER_ASSOCIATED_NAMES`.

ServiceListInfo

```
struct ServiceListInfo {
      wstring<64>     title;
      ServiceListType type;
      ushort          numEntries;
      uint            sizeHint;
      boolean         userOrdered;
};
```

Description

Each service list has a descriptive string – the title – which may be used when displaying the list. This string is constructed by the tuner (or Tuner FCM) in a proprietary manner. The way the list is ordered can be used for obtaining a consistent behavior of “channel-up/down” functionality (provided by applications) between different tuners. In addition, a service list has a `ServiceListType`, as described above, an indication of the number of entries, and a value called `sizeHint` which gives an upper bound on the number of bytes needed to store the service list. `sizeHint` of a service list of type `USER_ASSOCIATED_NAMES` should be the maximum number of bytes that the tuner allows users to set (if `Tuner::SetServiceList` is supported). The `userOrdered` field indicates whether the order of entries in the list has been determined by the user (`True`) or tuner/provider (`False`).

ServiceLocator

```
typedef sequence<octet, 256> ServiceLocator;
```

Description

Services, and components of services, are associated with an opaque `ServiceLocator` assigned by the tuner or Tuner FCM. It is recommended that service locators be persistent, i.e., a `ServiceLocator` value should select the same service regardless of re-powering the tuner or reinstalling the Tuner FCM. Construction of service locators is proprietary to the tuner or Tuner FCM; there is no guarantee that a service locator obtained from one Tuner FCM will be resolvable by another Tuner FCM even when the targeted tuners deal with the same type of broadcast.

Within the context of the Tuner FCM from which it was obtained, a `ServiceLocator` acts as a unique key for a service. If a service appears in several lists provided by the FCM, or several times in the same list, it will have the same locator.

Service

```
struct Service {
      ServiceLocator   locator;
      StreamType       type;
      octet[3]         language;
      wstring<32>      name;
};
```

Element	Description
locator	A value to uniquely identify the service within a Tuner FCM.
type	Categorization of the service. For example, DIGITAL_VIDEO__MPEG2_MP_ML indicates a MPEG2 video stream while MULTIPLEX__DVB indicates a collection of streams such as a transport multiplex or a program multiplex. If the Tuner FCM is unable to determine a value for type.maxBandwidth it shall set this field to 0.
language	If the service is associated with a particular language (e.g., audio or subtitles) this field contains an ISO-639 three character code. If the service is not associated with a particular language (e.g., video or a multiplex containing several audio components) the field may be empty.
name	A descriptive name of the service.

Services are used as entries in service lists, but an entry in a service list can also be empty. Empty entries can be used to facilitate index-based selection of services in an interoperable way on different tuners. For example, a user may construct matching service lists on two different tuners by inserting empty entries to indicate services one tuner can receive but the other cannot. Another use of this facility is in “channel-based” service selection for skipping “un-receivable” channels.

```
union ServiceListEntry switch(boolean){
   case True:    Service service;
   case False:   ; // empty entry in service list
```

};

MuxAction

```
enum MuxAction { APPEND, REPLACE, REMOVE, CLEAR };
```

Specifies the multiplexer action to be carried out when a service is selected.

ServiceEvent

```
struct ServiceEvent {
      ServiceEventType   type;
      wstring<64>        eventName;
      DateTime           startTime;
      DateTime           stopTime;
      wstring<32>        eventCategory;
      wstring<256>       eventDescription;
};
```

Element	Description
eventName	A descriptive name of the service event.

startTime	Start date and time of the service event. The DateTime fields are set to “ignored” if this information is unavailable.
stopTime	End date and time of the service event. The DateTime fields are set to “ignored” if this information is unavailable.
eventCategory	Text containing category description (e.g., “sports”, “movie, “news”). An empty string if unavailable.
eventDescription	Text containing extended description. An empty string if unavailable.

ServiceEventType

```
enum ServiceEventType {
        PROGRAM_EVENT,
        AV_EVENT,
        AUDIO_EVENT,
        DATA_EVENT,
        UNKNOWN_EVENT,
};
```

Element	Description
PROGRAM_EVENT	The event name identifies a broadcast program such as a television or radio program. Start and stop times indicate the duration of the program. Contents of the event description are unspecified but could include names of actors, director, date of production, plot outline etc.
AV_EVENT	The event name identifies an AV clip such as a piece of video being shown in a television news program. Start and stop times indicate the duration of the broadcast of the AV clip. Contents of the event description are unspecified but could include date and location of the clip etc.
AUDIO_EVENT	The event name identifies an audio clip such as a piece of music. Start and stop times indicate the duration of the broadcast of the audio clip. Contents of the event description are unspecified but could include names of musicians, composer, recording label, etc.
DATA_EVENT	The event name identifies a data broadcast. Start and stop times indicate the duration of the broadcast. Contents of the event description are unspecified.
UNKNOWN_EVENT	Contents of the event name, start and stop time, and description are completely unspecified.

ServiceEventPeriod

```
enum ServiceEventPeriod{
                    EV_CURRENT,
                    EV_CURRENTNEXT,
                    EV_TODAY,
                    EV_WEEK,
                    EV_ALL,
                };
```

Element	Description
EV_CURRENT	Current event information for a service.
EV_CURRENTNEXT	Current event information and next event information for a service.
EV_TODAY	Event information for the current date for a service.

EV_WEEK	Event information for the week starting at the current date for a service.
EV_ALL	All available event information for a service.

TunerCapability

```
enum TunerCapability  {
      SET_SERVICE_LIST,
      GET_SERVICE_COMPONENTS,
      GET_SERVICE_EVENTS_CURRENT,
      GET_SERVICE_EVENTS_CURRENT_NEXT,
      GET_SERVICE_EVENTS_TODAY,
      GET_SERVICE_EVENTS_WEEK,
      GET_SERVICE_EVENTS_ALL,
};
```

6.2.3 Tuner API

Tuner::GetServiceListInfo

Prototype

```
Status Tuner::GetServiceListInfo(
      out sequence<ServiceListInfo> list)
```

Parameters

- `list` – a list of service list descriptors. The safe parameter size limit is 16 `ServiceListInfo` values.

`GetServiceListInfo` returns a list of service list descriptors (`ServiceListInfo` values) for the service lists available from a Tuner FCM. The returned list shall contain at least one service list descriptor.

Tuner::GetServiceList

Prototype

```
Status Tuner::GetServiceList(
      in ushort listNumber,
      in ushort first,
      in ushort entries,
      out sequence<ServiceListEntry> list)
```

Parameters

- `listNumber` – specifies the list number of the service list to be returned.
- `first` – specifies the number of the first entry to read (0 indicates first of the list).
- `entries` – specifies the number of entries to read (0 indicates all).
- `list` – a list of services. The safe parameter size limit is 4 kBytes.

Description

This API returns the service list indicated by `listNumber`. This value is an index into the sequence returned by `GetServiceListInfo`, so, for example, the `listNumber` value of 0 would obtain the first service list. The locators within the service list can be used when issuing the `Tuner::SelectService` API. Within a service list of type `TUNER_ASSOCIATED_NAMES` or

PROVIDER_ASSOCIATED_NAMES, ServiceLocator values are unique. Components of services may appear in a service list.

The list may be read in chunks by using the parameters first and entries. In this case, list contains only the entries specified by the range first and entries. The API may return fewer entries than requested if the request exceeds the end of the complete list or the chunk size exceeds the safe parameter size.

If the Tuner FCM does not have the capability to provide a service list, it shall always return an empty service list (*list* contains no entries) with SUCCESS. In this case GetServiceListInfo returns *ServiceListInfo* of the empty service list. On the other hand, if the Tuner FCM has the capability to provide a service list, it will return either a non-empty service list with SUCCESS, or an empty service list with ELIST when the service list is not available.

Error codes

- Tuner::ELIST – if the Tuner FCM cannot currently provide a service list for the specified list number.

Tuner::SetServiceList

Prototype

```
Status Tuner::SetServiceList(
      in ushort listNumber,
      in sequence<ServiceListEntry> list)
```

Parameters

- listNumber – specifies the list number of the service list to be filled.
- list – a list of services. The safe parameter size limit is the sizeHint value associated with the service list (obtained via GetServiceListInfo).

Description

This API fills a service list indicated by listNumber, i.e. it replaces the existing contents with the specified contents. As a side-effect, this API also updates the numEntries field and userOrdered field (set to True) of the associated ServiceListInfo for the service list.

The value of listNumber is an index into the sequence returned by GetServiceListInfo, so, for example, the listNumber value of 0 would specify the first service list. The service list shall be of type USER_ASSOCIATED_NAMES. The services within the service list can be constructed from other service lists obtained via the Tuner::GetServiceList API. Components of services may appear in a service list.

Error codes

- ENOT_IMPLEMENTED – if the Tuner FCM does not support setting a service list.
- Tuner::ELIST – if the Tuner FCM cannot provide a service list for the specified list number.
- Tuner::ELIST_TYPE – if the service list for the specified list number is not of type USER_ASSOCIATED_NAMES.
- Tuner::ESERVICE – if one or more of the services in the service list for the specified list number are not correct (i.e. locator, type or language cannot be handled by the tuner).

Tuner::GetService

Prototype

```
Status Tuner::GetService(
      in ServiceLocator locator,
      out Service service)
```

Parameters

- `locator` – service for which stream type information is required.
- `service` – the `Service` structure for the specified service.

Description

This API returns the `Service` for a specific locator. This API can be used to determine information about locators which have not been obtained from service lists. (For example, locators obtained via `GetServiceComponents`, `GetSelectedServices`, or the `TunerServiceChanged` event.) The resulting value of `service.name` is unspecified (i.e., there is no implicit reference to a default service list). It is allowable that `locator` not appear on any service list provided by the Tuner FCM (for example, `locator` refers to a service component and the Tuner FCM does not include components in its service lists). However if the Tuner FCM cannot resolve `locator` then an error is returned.

Error codes

- `Tuner::ELOCATOR` – if the Tuner FCM cannot resolve the locator.

Tuner::GetServiceComponents

Prototype

```
Status Tuner::GetServiceComponents(
      in ServiceLocator locator,
      out sequence<ServiceLocator> list)
```

Parameters

- `locator` – service for which component information is required.
- `list` – a list of locators of service components. The safe parameter size limit is 16 `ServiceLocator` values.

Description

A service may consist of several components, for example a video stream and several audio streams for multiple languages. This API returns a list of components comprising a particular service available from the tuner. The locators within the list can be used when issuing the `Tuner::SelectService` API. If `locator` has no components (and the Tuner FCM supports service components) an empty list is returned.

Error codes

- `ENOT_IMPLEMENTED` – if the Tuner FCM does not support service components.
- `Tuner::ELOCATOR` – if the Tuner FCM cannot resolve the locator.
- `Tuner::EUNAVAILABLE` – if the Tuner FCM cannot currently retrieve service components for the locator.

Tuner::GetServiceEvents

Prototype

```
Status Tuner::GetServiceEvents(
      in ServiceLocator locator,
      in ServiceEventPeriod eventPeriod,
      out sequence<ServiceEvent> list)
```

Parameters

- `locator` – a service locator identifying the service for which event information is required.
- `eventPeriod` – duration of event information requested.
- `list` – a list of service events. The safe parameter size limit is 8 `ServiceEvent` values if `eventPeriod` is `EV_CURRENT`, 16 `ServiceEvent` values if `eventPeriod` is `EV_CURRENTNEXT`, 128 values if `eventPeriod` is `EV_TODAY`, 1024 values if `eventPeriod` is `EV_WEEK` and 8192 values if `eventPeriod` is `EV_ALL`.

Description

This API returns a list of events carried by a particular service available from the tuner. These lists are for information only; no selection is possible on an event. If `locator` has no events (and the Tuner FCM supports service events) an empty list is returned.

Error codes

- `ENOT_IMPLEMENTED` – if the Tuner FCM does not support service events.
- `Tuner::EPERIOD` – the tuner does not support service events for this `eventPeriod` (but does for some other period as indicated by `Tuner::GetCapability`).
- `Tuner::ELOCATOR` – if the Tuner FCM cannot resolve the locator.
- `Tuner::EUNAVAILABLE` – if the Tuner FCM cannot currently retrieve events for the locator.

Tuner::SelectService

Prototype

```
Status Tuner::SelectService(
      in ServiceLocator locator,
      in ushort plugNum, in MuxAction action)
```

Parameters

- `locator` – specifies the service to be selected.
- `plugNum` – the number of an output plug for the FCM.
- `action` – specifies the action to be carried out on the output plug. The following values can be set:
  `APPEND` – add (multiplex) the specified service to the output plug.
  `REPLACE` – remove all current services from the output plug, and output the specified service.
  `REMOVE` – remove the specified service from the output plug.
  `CLEAR` – stop the output of all services on the specified plug. In this case the `locator` parameter will be ignored.

Description

This API selects the service, which is identified by locator, and outputs (appends or removes, etc.) it to the specified plug. This API returns after the tuner has been configured according to the arguments given, regardless of whether or not the tuner has actually started to send the selected service.

Once this call completes successfully, `Tuner::GetSelectedServices` will return values appropriate to the tuner's new configuration.

For `REMOVE`, the locator may refer to a service present on the plug or to a component of a service present on the plug.

Selecting a service for one plug may preempt a service being delivered to another plug, however it is recommended that Tuner FCM implementations attempt to minimize such preemption. For example, suppose an AV multiplex is selected for *plug1*, the audio component selected for *plug2*, and then a different AV multiplex is selected for *plug1*. At least in the case that both AV multiplexes belong to the same parent multiplex (e.g. the same DVB multiplex), it is preferable that internal operation of the tuner be maintained so that the audio component of the second AV multiplex is output on *plug2*.

`Tuner::SelectService` may change the stream type associated with plugs (as determined via `Dcm::GetStreamType`) and so result in a `StreamTypeChanged` event.

Error codes

- `Tuner::ELOCATOR` – if the Tuner FCM cannot resolve the locator.
- `Tuner::EPLUG` – the plug does not exist.
- `Tuner::EREMOVE` – if the `REMOVE` action is requested when the service is not present on the specified plug.
- `Tuner::ENOT_COMPAT` – the plug cannot accept the services that would result from an `APPEND`, `REPLACE` or `REMOVE` action.

Tuner::GetSelectedServices

Prototype

```
Status Tuner::GetSelectedServices(
      in ushort plugNum, out sequence<ServiceLocator> list)
```

Parameters

- `plugNum` – the number of an FCM output plug.
- `list` – a list of services currently being output. The safe parameter size limit is 16 `ServiceLocator` values.

Description

This API returns a list of services currently being output via the specified plug.

If, through a series of service selections, all the components of a particular service are present on a plug then `list` shall contain the locator of the service itself and not the locators of the components.

Error codes

- `Tuner::EPLUG` – the plug does not exist.

Tuner::GetCapability

Prototype

```
Status Tuner::GetCapability(
      out sequence<boolean> capabilityList)
```

Parameters

- `capabilityList` – a list of capabilities supported by the Tuner FCM. The i^{th} member of `capabilityList` indicates the availability of the capability identified by `TunerCapability` value *i*. The safe parameter size is 32 `boolean` values.

Description

This API returns the capabilities of the Tuner FCM.

6.2.4 Tuner Events

TunerServiceChanged

Prototype

```
void TunerServiceChanged(
      in ushort plugNum,
      in sequence<ServiceLocator> list)
```

Parameters

- `plugNum` – the number an FCM output plug.
- `list` – a list of services currently being output (after a change occurs). The safe parameter size limit is 16 `ServiceLocator` values.

Description

This API notifies a change in services present on the output plug indicated by the `plugNum` parameter.

6.3 VCR FCM

6.3.1 VCR Services

Service	Comm Type	Locality	Access	Resv Prot
Vcr::Play	M	global	all	yes
Vcr::Record	M	global	all	yes
Vcr::FastForward	M	global	all	yes
Vcr::FastReverse	M	global	all	yes
Vcr::VariableForward	M	global	all	yes
Vcr::VariableReverse	M	global	all	yes
Vcr::Stop	M	global	all	yes
Vcr::RecPause	M	global	all	yes
Vcr::Skip	M	global	all	yes
Vcr::EjectMedia	M	global	all	yes
Vcr::GetState	M	global	all	
Vcr::GetRecordingMode	M	global	all	
Vcr::SetRecordingMode	M	global	all	yes
Vcr::GetFormat	M	global	all	
Vcr::GetPosition	M	global	all	
Vcr::ClearRTC	M	global	all	yes
Vcr::GetCapability	M	global	all	
Vcr::GetRejectInfo	M	global	all	
VcrStateChanged	E	global	VCR (all)	

6.3.2 VCR Data Structures

VcrRecordingMode

```
enum VcrRecordingMode {
      SPEED_SP,
      SPEED_LP,
      SPEED_EP,
      SPEED_VP
};
```

- `SPEED_SP` – “standard play” recording mode
- `SPEED_LP` – “long play” recording mode
- `SPEED_EP` – “extended play” recording mode
- `SPEED_VP` – “very long play” recording mode. For example, VHS/S-VHS has VP recording mode for NTSC.

The `VcrRecordingMode` is specified by the track pitch used for recording.

VcrTransportMode

```
enum VcrTransportMode {
      PLAY, RECORD,
      FAST_FORWARD, FAST_REVERSE,
      VARIABLE_FORWARD, VARIABLE_REVERSE,
      STOP, RECPAUSE, SKIP, NO_MEDIA
};
```

VcrTransportState

```
enum TapePosition {
        BEGINNING_OF_TAPE,
        END_OF_TAPE,
        OTHER_POSITION,
        UNKNOWN,
};

union VcrTransportState switch (VcrTransportMode) {
      case PLAY:               ;
      case RECORD:             VcrRecordingMode   rmode;
      case FAST_FORWARD:       ;
      case FAST_REVERSE:       ;
      case VARIABLE_FORWARD:   ForwardSpeed    fspeed;
      case VARIABLE_REVERSE:   ReverseSpeed    rspeed;
      case STOP:               TapePosition    tapePos;
      case RECPAUSE:           ;
      case SKIP:               SkipDirection   skipDir;
      case NO_MEDIA:           ;
};
```

VcrCounterType

```
enum VcrCounterType {
      RELATIVE_TIME, RELATIVE_COUNT,
      ABSOLUTE_TIME, ABSOLUTE_COUNT,
};
```

- `RELATIVE_TIME` – a relative time counter value (HH:MM:SS:FF)
- `RELATIVE_COUNT` – a relative counter number (decimal value)
- `ABSOLUTE_TIME` – an absolute time counter value (HH:MM:SS:FF)
- `ABSOLUTE_COUNT` – an absolute track number (decimal value)

VcrCounterValue

```
union VcrCounterValue switch (VcrCounterType) {
      case RELATIVE_TIME:   TimeCode   relTime;
      case RELATIVE_COUNT:  long       relCount;
      case ABSOLUTE_TIME:   TimeCode   absTime;
      case ABSOLUTE_COUNT:  long       absCount;
};
```

VcrRejectCondition

```
enum VcrRejectCondition  {
      NO_CASSETTE,
```

```
        READ_ERR,
        WRITE_ERR,
        WRITE_PROTECTED,
        END_OF_TAPE,
        BEGINNING_OF_TAPE,
        LOCKED,
        NO_CONNECTION,
        CONDENSATION,
        TRANSITION_NOT_AVAILABLE,
        UNKNOWN,
        ABORTED
};
```

- NO_CASSETTE – no cassette is loaded
- READ_ERR – the cassette cannot be read (e.g., because of dust or a scratch)
- WRITE_ERR – the cassette cannot be written (e.g., because of dust or a scratch)
- WRITE_PROTECTED – a write-protected cassette is loaded
- END_OF_TAPE – the current position is the end of the tape
- BEGINNING_OF_TAPE – the current position is the beginning of the tape
- LOCKED – the VCR is “hold locked”. (Some portable mobile devices have a small mechanical toggle switch called a “hold lock switch”. While this switch is ON, i.e., the VCR is hold locked, the device is guarded against operations such as accidental power on when not in use, or interruption of play or record from accidental pressing of a front panel button or a GUI button.)
- NO_CONNECTION – no device connection exists to the sink FCM plug
- CONDENSATION – vapor condensation has formed
- TRANSITION_NOT_AVAILABLE – vendor dependent functionality that inhibits the control command (e.g., the transition to or from a RECORD mode from any transport state except STOP)
- UNKNOWN – an unknown condition exists
- ABORTED – the operation invoked is not completed as intended because some other operation is requested from front panel button, IR remote, or other application, etc.

VcrCapability

```
enum VcrCapability {
        SKIP,
        EJECT_MEDIA,
        SET_RECORDING_MODE,
        SPEED_EP,
        SPEED_LP,
        SPEED_SP,
        SPEED_VP,
        CLEAR_RTC
};
```

6.3.3 VCR API

Vcr::Play

Prototype

```
Status Vcr::Play()
```

Description

This API plays the data on the loaded tape at normal speed.

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.

Vcr::Record

Prototype

```
Status Vcr::Record()
```

Description

This API records data on the loaded tape according to the current recording mode.

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.

Vcr::FastForward

Prototype

```
Status Vcr::FastForward()
```

Description

This API simply fast-forwards the tape (without playing). This fast-forward action is continued until the next API such as `Vcr::Play` or `Vcr::Stop` is issued or the end of the tape is reached.

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.

Vcr::FastReverse

Prototype

```
Status Vcr::FastReverse()
```

Description

This API simply fast-rewinds the tape (without playing). This fast-reverse action is continued until the next API such as `Vcr::Play` or `Vcr::Stop` is issued or the beginning of the tape is reached.

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.

Vcr::VariableForward

Prototype

```
Status Vcr::VariableForward(in ForwardSpeed speed)
```

Parameter

- `speed` – forward speed mode

Description

This API plays the loaded tape back at variable-speeds according to `speed`. If the target device does not support the specified `FAST_FORWARD_x` (or `SLOW_FORWARD_x`), then the device shall interpret it as `FASTEST_FORWARD` (or `SLOWEST_FORWARD`).

This variable-forward action is continued until the next API such as `Vcr::Play` or `Vcr::Stop` is issued or the end of the tape is reached.

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.
- `EINVALID_PARAMETER` – `speed` contains an invalid value.

Vcr::VariableReverse

Prototype

```
Status Vcr::VariableReverse(in ReverseSpeed speed)
```

Parameter

- `speed` – reverse speed mode

Description

This API plays the loaded tape back at variable-speeds, in reverse, according to `speed`. The actual speeds encoded by the `speed` have the following restrictions. If the target device does not support the specified `FAST_REVERSE_x` (or `SLOW_REVERSE_x`), then the device shall interpret it as `FASTEST_REVERSE` (or `SLOWEST_REVERSE`).

This variable-reverse action is continued until the next API such as `Vcr::Play` or `Vcr::Stop` is issued or the beginning of the tape is reached.

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.
- `Vcr::ENOT_SUPPORTED` – the target device does not support the specified reverse speed mode which is any slow reverse or normal reverse.
- `EINVALID_PARAMETER` – `speed` contains an invalid value.

Vcr::Stop

Prototype

```
Status Vcr::Stop()
```

Description

This API stops all motion of the tape transport mechanism.

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via

Vcr::GetRejectInfo.

Vcr::RecPause

Prototype

```
Status Vcr::RecPause()
```

Description

This API pauses the recording operation. This pause action is stopped when the next API such as Vcr::Record or Vcr::Stop is issued.

Error codes

- Vcr::EREJECTED – the API could not be executed. Reject conditions may be retrieved via Vcr::GetRejectInfo.

Vcr::Skip

Prototype

```
Status Vcr::Skip(
      in SkipDirection direction,
      in SkipMode mode, in uint count)
```

Parameters

- direction – direction to skip
- mode – skip mode
- count – number of items to be skipped, in accordance with the mode parameter. If mode is TIME, the format of this parameter is the same as the counterValue parameter for the Vcr::GetPosition API (in the case that the counterType is RELATIVE_TIME or ABSOLUTE_TIME). See the Vcr::GetPosition API below.

Description

This API fast-forwards (fast-reverses) the tape in the skip-mode specified by the mode parameter, skipping the number of items specified by the count parameter.

The SUCCESS response for this API should be returned after completion of skip operation. If the skip operation is aborted before its completion because of user's operation through the front panel button, IR remote or the other application, this API returns VCR::EREJECTED and set the reject condition to ABORTED.

The VCR should be paused in playback mode immediately after advancing or reversing the transport mechanism as the result of invoking this API.

This API is optional. Its support can be verified by the Vcr::GetCapability API.

Error codes

- ENOT_IMPLEMENTED – the target device does not support skip capability.
- Vcr::EREJECTED – the API could not be executed. Reject conditions may be retrieved via Vcr::GetRejectInfo.
- Vcr::ENOT_SUPPORTED – the target device does not support the specified skip mode.
- EINVALID_PARAMETER – skipDirection, skipMode or count contain invalid values.

Vcr::EjectMedia

Prototype

```
Status Vcr::EjectMedia()
```

Description

This API ejects the currently loaded tape from the VCR.

Error codes

- `ENOT_IMPLEMENTED` – the target device does not support the eject capability.
- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.

Vcr::GetState

Prototype

```
Status Vcr::GetState(
      out VcrTransportState state)
```

Parameters

- `state` – status of the VCR's transport mechanism

Description

This API returns the current status of the VCR's transport mechanism (playing, recording, etc.)

Vcr::GetRecordingMode

Prototype

```
Status Vcr::GetRecordingMode(out VcrRecordingMode mode)
```

Parameters

- `mode` – recording mode

Description

This API returns the current recording mode. This recording mode applies to record operations started by the `Vcr::Record` API and set by the `Vcr::SetRecordingMode` API.

Vcr::SetRecordingMode

Prototype

```
Status Vcr::SetRecordingMode(in VcrRecordingMode mode)
```

Parameters

- `mode` – Specifies the recording mode

Description

This API sets the recording mode. This recording mode applies to record operations started by the `Vcr::Record` API.

This API is optional. Its support can be verified by the `Vcr::GetCapability` API.

Error codes

- `ENOT_IMPLEMENTED` – the target device does not support setting the recording mode.
- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.
- `Vcr::ENOT_SUPPORTED` – the target device does not support the requested recording mode.

Vcr::GetFormat

Prototype

```
Status Vcr::GetFormat(
      out MediaFormatId format,
      out WriteProtectStatus writeStatus)
```

Parameters

- `format` – the format of the tape loaded in the VCR. See Annex 11.10 for possible values
- `writeStatus` – write protect status of the tape loaded in the VCR

Description

This API returns information concerning the tape loaded in the VCR (tape type, whether the tape can be recorded to, etc.)

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.

Vcr::GetPosition

Prototype

```
Status Vcr::GetPosition(
      in VcrCounterType type, out VcrCounterValue value)
```

Parameters

- `type` – specifies the type of counter value to be returned
- `value` – the counter value returned

Description

This API returns the current value of the counter.

At least one of the counter types should be supported. If `type` is not supported then `Vcr::GetPosition` returns `Vcr::ENOT_SUPPORTED`.

Error codes

- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.
- `Vcr::ENOT_SUPPORTED` – the target VCR does not support the counter type.
- `EINVALID_PARAMETER` – `counterType` contains in invalid value.

Vcr::ClearRTC

Prototype

```
Status Vcr::ClearRTC()
```

Description

This API clears (sets to 0) the value of the relative (time) counter. Timer counters are optional, their support can be verified by the `Vcr::GetCapability` API.

Error codes

- `ENOT_IMPLEMENTED` – the target device does not support timer counters.
- `Vcr::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `Vcr::GetRejectInfo`.

Vcr::GetCapability

Prototype

```
Status Vcr::GetCapability(
      out sequence<boolean> capabilityList,
      out sequence<MediaFormatId> playFormats,
      out sequence<MediaFormatId> recordFormats)
```

Parameters

- `capabilityList` – a list of capabilities supported by the VCR Functional Component Module. The i^{th} member of `capabilityList` indicates the availability of the capability identified by `VcrCapability` value *i*. The safe parameter size is 32 `boolean` values.
- `playFormats` – the tape formats the VCR is capable of playing. The safe parameter size limit is 4 `MediaFormatId` values.
- `recordFormats` – the tape formats the VCR is capable of recording. The safe parameter size limit is 4 `MediaFormatId` values.

Description

This API returns the capabilities of a VCR Functional Component Module.

Vcr::GetRejectInfo

Prototype

```
Status Vcr::GetRejectInfo (
      out sequence<VcrRejectCondition> rejectedConditions,
      out OperationCode rejectedOpcode,
      out sequence<VcrRejectCondition> currentRejectConditions)
```

Parameters

- `rejectedConditions` – contains one or more conditions for which the latest `Vcr::EREJECTED` error occurred. It is independent of whether the conditions causing the error still exist or have disappeared. Only if no `Vcr::EREJECTED` error has occurred will the list be empty. The safe parameter size limit is *n* `VcrRejectCondition` values, where *n* is the number of members of `VcrRejectCondition`.
- `rejectedOpcode` – the operation code of the last invoked API that caused a `Vcr::EREJECTED` status. Undefined in case `rejectedConditions` is empty.

- `currentRejectConditions` – contains one or more conditions that currently exist that would cause one or more APIs to return `Vcr::EREJECTED`. If no reject conditions currently exist the list will be empty. The safe parameter size limit is *n* `VcrRejectCondition` values, where *n* is the number of members of `VcrRejectCondition`.

The possible reject conditions for each VCR API service are listed below.

Service	Possible Conditions
`Vcr::Play`	`NO_CASSETTE, READ_ERR, END_OF_TAPE, LOCKED, CONDENSATION, TRANSITION_NOT_AVAILABLE`
`Vcr::Record`	`NO_CASSETTE, WRITE_ERR, WRITE_PROTECTED, END_OF_TAPE, LOCKED, NO_CONNECTION, CONDENSATION, TRANSITION_NOT_AVAILABLE`
`Vcr::FastForward`	`NO_CASSETTE, END_OF_TAPE, LOCKED, CONDENSATION, TRANSITION_NOT_AVAILABLE`
`Vcr::FastReverse`	`NO_CASSETTE, BEGINNING_OF_TAPE, LOCKED, CONDENSATION, TRANSITION_NOT_AVAILABLE`
`Vcr::VariableForward`	`NO_CASSETTE, END_OF_TAPE, LOCKED, CONDENSATION, TRANSITION_NOT_AVAILABLE`
`Vcr::VariableReverse`	`NO_CASSETTE, BEGINNING_OF_TAPE, LOCKED, CONDENSATION, TRANSITION_NOT_AVAILABLE`
`Vcr::Stop`	`NO_CASSETTE, LOCKED`
`Vcr::RecPause`	`NO_CASSETTE, WRITE_PROTECTED, LOCKED, CONDENSATION, TRANSITION_NOT_AVAILABLE`
`Vcr::Skip`	`NO_CASSETTE, LOCKED, END_OF_TAPE, BEGINNING_OF_TAPE, CONDENSATION, TRANSITION_NOT_AVAILABLE, ABORTED`
`Vcr::EjectMedia`	`NO_CASSETTE, LOCKED, CONDENSATION, TRANSITION_NOT_AVAILABLE`
`Vcr::SetRecordingMode`	`LOCKED, TRANSITION_NOT_AVAILABLE`
`Vcr::GetFormat`	`CONDENSATION`
`Vcr::GetPosition`	`NO_CASSETTE, CONDENSATION`
`Vcr::ClearRTC`	`LOCKED`

Description

This API returns information on the conditions that caused or could cause `Vcr::EREJECTED` to be returned as a result of the VCR APIs.

6.3.4 VCR Events

VcrStateChanged

Prototype

```
void VcrStateChanged(
      in VcrTransportState state,
      in MediaFormatId format)
```

Parameters

- `state` – status of the VCR's transport mechanism
- `format` – format of tape currently loaded in the VCR

Description

This API notifies of changes in the state of a VCR's transport mechanism.

6.3.5 VCR Notification Attributes

Vcr::currentState

Attribute

```
struct VcrCurrentState {
      VcrTransportState  state;
      MediaFormatId      format;
} currentState
```

Description

New setting of `VcrTransportState`.

Vcr::recordingMode

Attribute

```
VcrRecordingMode mode
```

Description

New setting of recording mode.

Vcr::counterSet

Attribute

```
boolean counterSet    // the value is irrelevant
```

Description

Setting of the counter. Only useful with comparator `ANY`. Relates to reset (to zero) or setting of a specific value, e.g. via front panel operation. Normal increase or decrease of the counter is not notified.

Vcr::condensation

Attribute

```
boolean condensation
```

Description

Indicates whether vapor condensation has been detected in the VCR.

6.4 Clock FCM

6.4.1 Clock Services

Service	Comm Type	Locality	Access	Resv Prot
Clock::GetDateTime	M	global	all	
Clock::SetDateTime	M	global	all	
Clock::GetTimezone	M	global	all	
Clock::SetTimezone	M	global	all	
Clock::EnableAutoDST	M	global	all	
Clock::IsEnabledAutoDST	M	global	all	
Clock::GetCapability	M	global	all	
Clock::CreateTimer	M	global	all	
Clock::GetTimerState	M	global	all	
Clock::SetTimerState	M	global	all	
Clock::DeleteTimer	M	global	all	
<Client>::TimerFired	MB	global	Clock (all)	

The Clock FCM provides a set of APIs for querying clocks and setting their values. The APIs allow for clocks of varying capability. Examples of capabilities include self-power (e.g., battery powered clocks) and the ability to automatically reset.

Clocks are in one of two states: 1) normal running state, and 2) in need of reset. When in the running state the clock will return the current time (and possibly date and time zone). When in the "needs reset" state, the clock will not return the current time. Transition from "needs reset" to running occurs via `Clock::SetDateTime` or automatically if the clock has the capability to "lock" on to an externally provided time signal.

Some clocks may also support *timers*. Timers can be set to *fire* at a specified time and date. When a timer fires it sends a `<Client>::TimerFired` message to the SEID which set the timer.

6.4.2 Clock Data Structures

Timezone

```
struct Timezone {
   short   gmtOffset;    //in minutes
   boolean DST;
};
```

Time zones are represented by an offset, measured in minutes, from Greenwich Mean Time. The offset is positive to the east and negative to the west. To convert from GMT to the local time, the time zone is added, taking the sign of the offset into account. The `DST` flag indicates whether day light savings time is in effect or not. If times were measured in elapsed minutes from some reference time, the formula for converting from a GMT time to a local time would be: `local = gmt + gmtOffset + (DST * 60)`

ClockCapabilityStatus

```
enum ClockCapabilityStatus {
   NOT_SUPPORTED,
   MANUAL_UNSET,
   MANUAL_SET,
   AUTOMATIC
};
```

`ClockCapabilityStatus` values indicate the capability of the various date, time and time zone fields of a Clock FCM:

- `NOT_SUPPORTED` – the field is not supported by the Clock FCM
- `MANUAL_UNSET` – the field is manually settable but has not been set
- `MANUAL_SET` – the field is manually settable and has been set
- `AUTOMATIC` – the field is automatically set by hardware (or the Clock FCM), manual setting of the value may be overwritten by the automatic setting

ClockCapability

```
struct ClockCapability {
   ClockCapabilityStatus    year;
   ClockCapabilityStatus    month;
   ClockCapabilityStatus    day;
   ClockCapabilityStatus    dayOfWeek;
   ClockCapabilityStatus    hour;
   ClockCapabilityStatus    minute;
   ClockCapabilityStatus    sec;
   ClockCapabilityStatus    msec;
   ClockCapabilityStatus    timezone;
   ClockCapabilityStatus    DSTEnableStyle;
   boolean                  selfPowered;
};
```

A Clock FCM implementation is required to support hour and minute fields (i.e., the `ClockCapabilityStatus` values for these fields shall not be `NOT_SUPPORTED`).

The `selfPowered` flag indicates whether the clock will continue to run if the hosting device looses power.

TimerId

```
typedef ushort TimerId;
```

6.4.3 Clock API

Clock::GetDateTime

Prototype

```
Status Clock::GetDateTime(out DateTime dateTime)
```

Parameters

- `dateTime` – the `DateTime` value to which the clock is currently set

Description

`Clock::GetDateTime` obtains the current date and time from the clock. If all values of the `DateTime` structure are not set, this API will return `Clock::EUNSET`. If at least one value is set, `dateTime` will be returned with unset values set as "ignored".

Error codes

- `Clock::EUNSET` – the clock has not been set

Clock::SetDateTime

Prototype

```
Status Clock::SetDateTime(in DateTime dateTime)
```

Parameters

- `dateTime` – the `DateTime` value used to set the clock

Description

`Clock::SetDateTime` sets the current time of the clock. If a `DateTime` field within the clock was previously set and the corresponding input `DateTime` field is set as "ignored", the field will remain set with its previous value.

For the clock to be validly set, the following conditions must apply:

- All supported time fields (`hour`, `minute`, `sec`, `msec`) must become, or previously have been, set.
- If supported, the `year`, `month` and `day` fields may optionally be set, but they must become set as complete group. In this case the `dayOfWeek` field is ignored.
- If `dayOfWeek` is supported, and the `year`, `month` and `day` are not set, then the `dayOfWeek` field may be set.

If the clock does not become validly set, `Clock::ESET` is returned.

Changing the `dateTime` of the Clock may put the `fireTime` of one or more of the timers into the past. All timers with their `fireTime` put into the past will be fired immediately.

Error codes

- `Clock::ESET` – invalid `DateTime` value

Clock::GetTimezone

Prototype

```
Status Clock::GetTimezone(out Timezone myZone)
```

Parameters

- `myZone` – the time zone currently used by the clock

Description

`Clock::GetTimezone` obtains the current time zone from the clock.

Error codes

- `Clock::EZONE` – the clock does not support time zones
- `Clock::EUNSET` – the time zone has not been set

Clock::SetTimezone

Prototype

```
Status Clock::SetTimezone(in Timezone myZone)
```

Parameters

- `myZone` – the `Timezone` value used to set the clock

Description

`Clock::SetTimezone` sets the time zone of the clock. Note that the day light savings time can be turned on and off by using `Clock::GetTimezone` followed by `Clock::SetTimezone` with the `DST` flag switched.

Error codes

- `Clock::EZONE` – the clock does not support time zones
- `Clock::ESET` – invalid `Timezone` value

Clock::EnableAutoDST

Prototype

```
Status Clock::EnableAutoDST(in boolean enable)
```

Parameters

- `enable` – a boolean flag

Description

`Clock::EnableAutoDST` controls whether or not the clock will perform automatic DST adjustment. This request will change the status of `ClockCapability.DSTEnableStyle` to `AUTOMATIC` if `enabled` is `True` and to `MANUAL_SET` if `enabled` is `False`.

Error codes

- `Clock::EAUTO_DST` – the clock does not support auto DST

Clock::IsEnabledAutoDST

Prototype

```
Status Clock::IsEnabledAutoDST(out boolean isEnabled)
```

Parameters

- `isEnabled` – a boolean flag

Description

`Clock::IsEnabledAutoDST` determines whether or not automatic DST adjustment is enabled. The value of `isEnabled` is `True` if `ClockCapability.DSTEnableStyle` is `AUTOMATIC` and `False` otherwise.

Error codes
- `Clock::EAUTO_DST` – the clock does not support auto DST

Clock::GetCapability

Prototype
```
Status Clock::GetCapability(out ClockCapability capability)
```

Parameters
- `capability` – a `ClockCapability` value

Description

`Clock::GetCapability` returns the basic capabilities of the clock.

Clock::CreateTimer

Prototype
```
Status Clock::CreateTimer(out TimerId timer)
```

Parameters
- `timer` – id of a timer

Description

`Clock::CreateTimer` returns the `TimerId` of an unused timer.

Error codes
- `ENOT_IMPLEMENTED` – if the target device does not support timers
- `Clock::ENO_FREE` – no timers are available

Clock::GetTimerState

Prototype
```
Status Clock::GetTimerState(
      in TimerId timer,
      out OperationCode opCode,
      out DateTime fireTime,
      out uint fireVal,
      out SEID owner)
```

Parameters
- `timer` – a timer id
- `opCode` – the operation code used to notify the client via `<Client>::TimerFired`
- `fireTime` – the date and time the timer will fire
- `fireVal` – the value passed in `<Client>::TimerFired` when the timer fires
- `owner` – who set the timer

Description

`Clock::GetTimerState` obtains the data associated with a timer.

Error codes

- `ENOT_IMPLEMENTED` – if the target device does not support timers
- `Clock::ETIMER` – `timer` is not a valid `TimerId`
- `Clock::EUNSET` – the timer is not set

Clock::SetTimerState

Prototype

```
Status Clock::SetTimerState(
      in TimerId timer,
      in OperationCode opCode,
      in DateTime fireTime,
      in uint fireVal)
```

Parameters

- `timer` – a timer id
- `opCode` – the operation code used to notify the client via `<Client>::TimerFired`
- `fireTime` – the date and time the timer will fire
- `fireVal` – the value passed in `<Client>::TimerFired` when the timer fires

Description

`Clock::SetTimerState` sets the data associated with a timer.

In order to set the timer, all supported `DateTime` fields must become, or previously have been, set. Otherwise `Clock::ESET` is returned.

If `fireTime` is set to a time which has already passed, the timer will fire immediately.

Error codes

- `ENOT_IMPLEMENTED` – if the target device does not support timers
- `Clock::ETIMER` – `timer` is not a valid `TimerId`
- `Clock::ESET` – invalid `DateTime` value

Clock::DeleteTimer

Prototype

```
Status Clock::DeleteTimer(in TimerId timer)
```

Parameters

- `timer` – a timer id

Description

`Clock::DeleteTimer` deletes a timer which is in use. After a timer has been deleted, it shall not call `<Client>::TimerFired`.

Error codes

- `ENOT_IMPLEMENTED` – if the target device does not support timers
- `Clock::ETIMER` – `timer` is not a valid `TimerId`
- `Clock::ENOT_OWNER` – the timer is set by another SEID than the caller

<Client>::TimerFired

Prototype

```
Status  <Client>::TimerFired(
      in SEID clock,
      in TimerId timer,
      in uint fireVal)
```

Parameters

- `clock` – the SEID of the Clock FCM on which the timer resides
- `timer` – a timer id
- `fireVal` – the value specified via `Clock::SetTimerState`

Description

A `<Client>::TimerFired` message is sent by a Clock FCM to the SEID that has set a timer. The operation code used by the Clock FCM to send this message is that provided by the client when it invoked `Clock::SetTimerState`. The `TimerFired` message is sent when the timer fires, the timer then becomes free. No event or error is generated by the Clock FCM if message delivery fails (such as when the target SEID is no longer present).

6.4.4 Clock Notification Attributes

Clock::dateTime

Attribute

```
DateTime dateTime
```

Description

New setting of date and/or time.

Clock::timezone

Attribute

```
Timezone timezone
```

Description

New setting of timezone.

Clock::DSTEnabled

Attribute

```
boolean DSTEnabled
```

Description

New setting of DST: `True` if enabled, `False` if disabled.

6.5 Camera FCM

The HAVi Camera FCM enables an application to control camera functional components located in devices connected to HAVi network. It provides APIs required to control the camera functional components with minimal interoperability between various cameras from different manufacturers.

Scope – The HAVi Camera FCM supports both standalone camera devices and camera functional components inside devices. Examples are the camera portion of camcorders, digital still cameras, camera devices for of surveillance, and so on. These cameras are classified into "video cameras" and "still cameras." A video camera is a device that can be a source of video stream. It takes images by its CCD and outputs a raw or compressed video stream with some kind of video signal processing. A still camera is a device that can capture and store still images. The stored images can be accessed from the application. Moreover, devices exist which support both the video camera functionality and the still camera functionality.

The HAVi FCM covers functionality of both video cameras and still cameras.

Note – The current version of HAVi specification does not define APIs for file I/O or transferring non-streaming data such as still images stored in a camera in an isochronous way from one FCM to another. For this reason the HAVi Camera FCM provides APIs for transferring image data from the FCM to another software element.

6.5.1 Camera Services

Service	Comm Type	Locality	Access	Resv Prot
Camera::Zoom	M	global	all	yes
Camera::Pan	M	global	all	yes
Camera::Tilt	M	global	all	yes
Camera::SetVideoState	M	global	all	yes
Camera::GetVideoState	M	global	all	
CameraVideoStateChanged	E	global	Camera (all)	
Camera::Shoot	M	global	all	yes
Camera::GetImageList	M	global	all	
Camera::OpenImage	M	global	all	
Camera::ReadImage	M	global	all	
Camera::CloseImage	M	global	all	
Camera::EraseImage	M	global	all	yes
Camera::GetCapability	M	global	all	

6.5.2 Camera Data Structures

ZoomOperation

```
enum ZoomOperation { TELE, WIDE, STOP };
```

- `TELE` – starts to change zoom factor to get a larger image
- `WIDE` – starts to change zoom factor to get a smaller image

- `STOP` – stop zoom

PanOperation

```
enum PanOperation { LEFT, RIGHT, STOP };
```

- `LEFT` – starts to pan camera to left
- `RIGHT` – starts to pan camera to right
- `STOP` – stop pan

TiltOperation

```
enum TiltOperation { UP, DOWN, STOP };
```

- `UP` – starts to tilt camera upwards
- `DOWN` – starts to tilt camera downwards
- `STOP` – stop tilt

StoredImage

```
struct StoredImage {
      ushort        index;
      wstring<12>   imageString;
      ImageTypeId   imageTypeId;
      ulong         imageSize;
};
```

- `index` – unique number that identifies an image stored in the camera
- `imageString` – contains a string intended to describe the image. It is supplied in a proprietary way.
- `imageTypeId` – one of the values given in Annex 11.13. It describes the type of image.
- `imageSize` – size in bytes of the stored image

CameraCapability

```
enum CameraCapability {
      ZOOM,
      PAN,
      TILT,
      VIDEO,
      STILL,
      SET_VIDEO_STATE
   };
```

6.5.3 Camera API

Camera::Zoom

Prototype

```
Status Camera::Zoom(in ZoomOperation zoom)
```

Parameters

- `zoom` – denotes the direction of change of the zoom factor

Description

`Camera::Zoom` provides application control of the zoom factor of image being shot or to be shot. The parameter `zoom` denotes the actual zoom control to be performed. Issuing `Zoom(TELE)` causes the camera to change zoom factor continuously in the direction that makes the image larger until `Zoom(STOP)` is issued, while `Zoom(WIDE)` causes the camera to change zoom factor to make the image smaller. This API returns `ENOT_IMPLEMENTED` if the corresponding capability is not supported.

Error codes

- `ENOT_IMPLEMENTED` – The target device or FCM does not have the zoom capability.
- `Camera::EREJECTED` – The target device could not execute the zoom operation.
- `EINVALID_PARAMETER` – The value of `zoom` is invalid.

Camera::Pan

Prototype

```
Status Camera::Pan(in PanOperation pan)
```

Parameters

- `pan` – denotes the direction of change of the pan

Description

`Camera::Pan` provides application control of the horizontal angle of the camera. The parameter `pan` denotes the actual pan control to be performed. Issuing `Pan(LEFT)` causes the camera to move or rotate to the left until `Pan(STOP)` is issued, while `Pan(RIGHT)` causes it to move to the right. Note that the images on the display screen move in the opposite direction to the parameter `pan`. This API returns `ENOT_IMPLEMENTED` if the corresponding capability is not supported.

Error codes

- `ENOT_IMPLEMENTED` – The target device or FCM does not have the pan capability.
- `Camera::EREJECTED` – The target device could not execute the pan operation.
- `EINVALID_PARAMETER` – The value of `pan` is invalid.

Camera::Tilt

Prototype

```
Status Camera::Tilt(in TiltOperation tilt)
```

Parameters

- `tilt` – denotes the direction of change of the tilt

Description

`Camera::Tilt` provides application control of the vertical angle of the camera. The parameter `tilt` denotes the actual tilt control to be performed. Issuing `Tilt(UP)` causes the camera to move upwards until `Tilt(STOP)` is issued, while `Tilt(DOWN)` causes it to move downwards. Note that the images on the display screen move in the opposite direction to the parameter `tilt`. This API returns `ENOT_IMPLEMENTED` if the corresponding capability is not supported.

Error codes

- `ENOT_IMPLEMENTED` – The target device or FCM does not have the tilt capability.

- `Camera::EREJECTED` – The target device could not execute the tilt operation.
- `EINVALID_PARAMETER` – The value of `tilt` is invalid.

Camera::SetVideoState

Prototype
```
Status Camera::SetVideoState(inout boolean videoOn)
```

Parameters
- `videoOn` – identifies whether video is output or not

Description

`Camera::SetVideoState` provides application control of the camera's video output. The parameter `videoOn` denotes the application's desired state of video output. Issuing `SetVideoState(True)` causes the camera to send its video. Issuing `SetVideoState(False)` turns the video off. This API returns `ENOT_IMPLEMENTED` for target devices or FCMs which do not have the `VIDEO` camera capability. If a video camera does not support video output on/off functionality then this API returns `ENOT_IMPLEMENTED`.

Error codes
- `ENOT_IMPLEMENTED` – The target device or FCM does not have the `VIDEO` camera capability (as indicated by `Camera::GetCapability`) or it does not support video output on/off functionality.
- `Camera::EREJECTED` – The target device could not change the state of video output. (This error code shall not be returned if `videoOn` is `True` and video output is already on.)

Camera::GetVideoState

Prototype
```
Status Camera::GetVideoState(out boolean videoOn)
```

Parameters
- `videoOn` – identifies whether video output is on

Description

`Camera::GetVideoState` enables an application to know the current state of camera's video output. The parameter `videoOn` denotes the current state of the video output. This API returns `ENOT_IMPLEMENTED` for target devices or FCMs which do not have the `VIDEO` camera capability.

Error codes
- `ENOT_IMPLEMENTED` – The target device or FCM does not have the `VIDEO` camera capability (as indicated by `Camera::GetCapability`).

Camera::Shoot

Prototype
```
Status Camera::Shoot()
```

Description

This API is only valid for target devices or FCMs which have the `STILL` camera capability. Issuing `Camera::Shoot` causes the camera to shoot an image and store it in the local storage media.

Error codes

- `ENOT_IMPLEMENTED` – The target device or FCM does not have the `STILL` camera capability (as indicated by `Camera::GetCapability`).
- `Camera::EREJECTED` – The target device could not execute the shoot operation.

Note

Note that for a camcorder to record video onto its tape, the appropriate VCR and Stream Manager APIs must be used to connect the Camera FCM and VCR FCMs. In case the video state of the Camera FCM is `False`, it is left to the implementation what will be recorded on tape (black screen, random data, …)

Camera::GetImageList

Prototype

```
Status Camera::GetImageList(out sequence<StoredImage> ilist)
```

Parameters

- `ilist` – a list containing indexes of stored images and other information. The safe parameter size limit is 50 `StoredImage` values.

Description

`Camera::GetImageList` enables an application to get information necessary to manipulate the images stored in the camera's local storage media. The parameter `ilist` contains information about each image such as file name or number, the data format, the size, and its index number. This API is only valid for target devices or FCMs which have the `STILL` camera capability.

The image data stored in the camera may change when new images are shot or old images are deleted, whether by `Camera::Shoot` or `Camera::EraseImage`, or by the user's direct operation to the device. To avoid having applications read or erase images other than intended due to inconsistency of the index number of images between applications, the FCM shall not reuse the index value of erased images for new images shot until all the index values which can be represented by a 16-bit `ushort` integer have run out.

Example:

index	image string	ImageTypeId	size (in Bytes)
1	aaa.jpg	0x0005	50,000
2	bbb.jpg	0x0005	60,000
5	ccc.bmp	0x0007	300,000

Error codes

- `ENOT_IMPLEMENTED` – The target device or FCM does not have the `STILL` camera capability (as indicated by `Camera::GetCapability`).
- `Camera::EREJECTED` – The target device could not send the image list.

Camera::OpenImage

Prototype

```
Status Camera::OpenImage(
      in ushort index,
      out short iHandle
   )
```

Parameters

- `index` – index of stored image to be opened.
- `iHandle` – denotes stored image to be selected.

Description

`Camera::OpenImage` is used to establish a data transfer path from a Camera FCM to an application. The application calls this API and prepares a data buffer for receiving image data. The image desired is specified by an index which can be obtained by `Camera::GetImageList` before using this API. The FCM returns the handle of opened image data which is used for actual data transfer operation by the `Camera::ReadImage` and `Camera::CloseImage` APIs. The handle is used in case of multiple application access to the same image at the same time, though it is not necessarily supported by all Camera FCMs. This API is only valid for target devices or FCMs which have the `STILL` camera capability.

Error codes

- `ENOT_IMPLEMENTED` – The target device or FCM does not have the `STILL` camera capability (as indicated by `Camera::GetCapability`).
- `Camera::EREJECTED` – The target device could not open the image.
- `EINVALID_PARAMETER` – The value of `index` is invalid, i.e. the target device does not have an image corresponding to the index.

Camera::ReadImage

Prototype

```
Status Camera::ReadImage(
      in short iHandle,
      in short datalength,
      out short dataleft,
      out sequence<octet> imagedata
   )
```

Parameters

- `iHandle` – denotes the image handle from which image data is read.
- `datalength` – the amount of image data to be transferred specified by the client.
- `dataleft` – the rest of image data to be transferred after this transfer.
- `imagedata` – denotes stored image to be selected. The safe parameter size limit is `datalength` bytes.

Description

`Camera::ReadImage` allows an application to receive data from the Camera FCM using the handle opened by `OpenImage` API. The `iHandle` value returned by `Camera::OpenImage` denotes the actual source of image data. The application calls `Camera::ReadImage` with the `iHandle` of the desired image, `datalength` specifies the length of data which is to be transferred

as a response message to this API call. The datalength should be determined by the application related to its buffer size. The Camera FCM sends back the image data, or possibly the part of image data that is requested by the ReadImage call. The amount of data that is specified by datalength parameter but could not be transferred in the response is returned as the out parameter dataleft. A dataleft value of zero indicates that all the image data specified by the API call has been transmitted. Consequently, in order to acquire an entire image, an application must call ReadImage until the sum of (datalength - dataleft) in each call reaches the imageSize of the desired image. This API is only valid for target devices or FCMs which have the STILL camera capability.

Error codes

- ENOT_IMPLEMENTED – The target device or FCM does not have the STILL camera capability (as indicated by Camera::GetCapability).
- Camera::EREJECTED – The target device is not ready to send a chunk of image data.
- EINVALID_PARAMETER – The value of iHandle or datalength is invalid.

Camera::CloseImage

Prototype

```
Status Camera::CloseImage(in short iHandle)
```

Parameters

- iHandle – denotes the image handle to be closed.

Description

Camera::CloseImage is used to terminate the data transfer from the Camera FCM to the Application. This API is only valid for target devices or FCMs which have the STILL camera capability.

Error codes

- ENOT_IMPLEMENTED – The target device or FCM does not have the STILL camera capability (as indicated by Camera::GetCapability).
- Camera::EREJECTED – The target device could not close the image.
- EINVALID_PARAMETER – The value of iHandle is invalid.

Camera::EraseImage

Prototype

```
Status Camera::EraseImage(in ushort index)
```

Parameters

- index – denotes stored image to be erased.

Description

Camera::EraseImage is used by an application to erase an image specified by its index in the camera device. The index has to be obtained by calling Camera::GetImageList before this API is used. This API is only valid for target devices or FCMs which have the STILL camera capability.

Error codes

- ENOT_IMPLEMENTED – The target device or FCM does not have the STILL camera capability (as indicated by Camera::GetCapability).

- `Camera::EREJECTED` – The target device could not erase the image.
- `EINVALID_PARAMETER` – The value of `index` is invalid, i.e. the target device does not have the image that corresponds to the index.

Camera::GetCapability

Prototype

```
Status Camera::GetCapability(
      out sequence<boolean> capabilityList)
```

Parameters

- `capabilityList` – a list of capabilities supported by the Camera FCM. The i^{th} member of `capabilityList` indicates the availability of the capability identified by `CameraCapability` value *i*. The safe parameter size is 32 `boolean` values.

Description

`Camera::GetCapability` returns the capability of the FCM. An application can get information about the specific Camera FCM with this API before using other APIs which the FCM does not support.

6.5.4 Camera Events

CameraVideoStateChanged

Prototype

```
void CameraVideoStateChanged(in boolean videoOn)
```

Parameters

- `videoOn` – identifies the current state of the video output

Description

`CameraVideoStateChanged` notifies an application of a change in state of camera's video output. The parameter `videoOn` denotes the current state of the video output.

6.5.5 Camera Notification Attributes

Camera::videoState

Attribute

```
boolean videoState
```

Description

Activation of video signal camera output. `True` if there is a video signal, `False` if there is no video signal.

Camera::zoom

Attribute

```
ZoomOperation zoom
```

Description

Setting of zoom operation.

Camera::pan

Attribute

```
PanOperation pan
```

Description

Setting of pan operation.

Camera::tilt

Attribute

```
TiltOperation tilt
```

Description

Setting of tilt operation.

6.6 AV Disc FCM

This section specifies APIs for the AV Disc FCM.

Note: This FCM API supports a set of common operations for isochronous data discs (AV Discs) only. When these APIs are used for a non-AV Disc (one which has a filesystem), they are not guaranteed or may be handled in a proprietary manner.

6.6.1 AV Disc Services

Service	Comm Type	Locality	Access	Resv Prot
AvDisc::GetItemList	M	global	all	
AvDisc::Play	M	global	all	yes
AvDisc::Record	M	global	all	yes
AvDisc::VariableForward	M	global	all	yes
AvDisc::VariableReverse	M	global	all	yes
AvDisc::Stop	M	global	all	yes
AvDisc::RecPause	M	global	all	yes
AvDisc::Skip	M	global	all	yes
AvDisc::InsertMedia	M	global	all	yes
AvDisc::EjectMedia	M	global	all	yes
AvDisc::GetState	M	global	all	
AvDisc::GetFormat	M	global	all	
AvDisc::GetPosition	M	global	all	
AvDisc::Erase	M	global	all	yes
AvDisc::PutItemList	M	global	all	yes
AvDisc::GetCapability	M	global	all	
AvDisc::GetRejectInfo	M	global	all	
AvDiscItemListChanged	E	global	AV Disc (all)	
AvDiscStateChanged	E	global	AV Disc (all)	

6.6.2 AV Disc Data Structures

ItemIndex

```
struct ItemIndex {
      ushort        list;
      ushort        index;
      wstring       title;
      wstring       artist;
      wstring       genre;
      wstring       contentType;
      TimeCode      playbackTime;
      ulong         contentSize;
      DateTime      initialTimeStamp;
```

```
      DateTime      lastUpdateTimeStamp;
};
```

Element	Description
list	List number to uniquely identify the child list. If the ItemIndex refers to a child list, the value indicates the list number of the child list and the ItemIndex contains the information of the child list. If the ItemIndex does not refer to a child list, the value is 0xFFFF and the ItemIndex contains the information of the track identified by index. The value corresponding to index=0 indicates the list that contains the ItemIndex itself and the ItemIndex contains the same information as the list. The value 0x0000 corresponding to index=0 always indicates the root list and the ItemIndex contains the information of the disc media.
index	The number of the index to uniquely identify the track.
title	Title of contents.
artist	Artist of contents.
genre	Genre of contents.
contentType	Type of contents(e.g. Video, Audio, Image, others).
playbackTime	Playback time of contents. The value corresponding to index=0 indicates the total playback time of disc. In case of a writable disc, it indicates the original time capacity of the disc media instead of total playback time.
contentSize	Playback byte size of contents. The value corresponding to index=0 indicates the total playback byte size of disc. In case of a writable disc, it indicates the original byte size capacity of the disc media instead of total playback byte size.
initialTimeStamp	The time when a content was first created on the disc media. The value corresponding to index=0 indicates the oldest time stamp on the disc.
lastUpdateTimeStamp	The time when a content was last updated on the disc media. The format of this parameter is the same as that of initialTimeStamp. The value corresponding to index=0 means the most recent time stamp on the disc.

AvDiscPlayMode

```
enum AvDiscPlayMode {
      NORMAL, DIRECT_1, DIRECT,
      REPEAT_1, REPEAT_ALL, SHUFFLE,
      RANDOM
};
```

AvDiscRecordingMode

```
enum AvDiscRecordingMode {
      NORMAL, NEW,
      OVERWRITE, OVERWRITE_AND_JUMP
```

```
};
```

AvDiscTransportMode

```
enum AvDiscTransportMode {
      PLAY, RECORD,
      VARIABLE_FORWARD, VARIABLE_REVERSE,
      STOP, RECPAUSE, SKIP, NO_MEDIA
};
```

AvDiscTransportState

```
union AvDiscTransportState switch (AvDiscTransportMode) {
      case PLAY:              AvDiscPlayMode        pmode;
      case RECORD:            AvDiscRecordingMode   rmode;
      case VARIABLE_FORWARD:  ForwardSpeed    fspeed;
      case VARIABLE_REVERSE:  ReverseSpeed    rspeed;
      case STOP:              ;
      case RECPAUSE:          ;
      case SKIP:              SkipDirection   skipDir;
      case NO_MEDIA:          ;
};
```

AvDiscCounterType

```
enum AvDiscCounterType {
      RELATIVE_TIME,
      ABSOLUTE_TIME,
      TRACK_NUMBER
};
```

- `RELATIVE_TIME` – relative time from the beginning of current track
- `ABSOLUTE_TIME` – absolute time from the beginning of first track (i.e., beginning of the disc)
- `TRACK_NUMBER` – current track

AvDiscCounterValue

```
union AvDiscCounterValue switch (AvDiscCounterType) {
      case RELATIVE_TIME:   TimeCode   relTime;
      case ABSOLUTE_TIME:   TimeCode   absTime;
      case TRACK_NUMBER:    long       track;
};
```

AvDiscCapability

```
enum AvDiscCapability {
      VARIABLE_FORWARD,
      VARIABLE_REVERSE,
      SKIP,
      REC_PAUSE,
      RECORD,
      GET_ITEM_LIST,
      PUT_ITEM_LIST,
      ERASE,
};
```

AvDiscRejectCondition

```
enum AvDiscRejectCondition {
      EMPTY_DISC,
      NO_DISC,
      SHORT_CAPACITY,
      READ_ERR,
      WRITE_ERR,
      WRITE_PROTECTED,
      DISC_FULL,
      TRACK_FULL,
      TOC_EDIT_BUSY,
      LOCKED,
      NO_CONNECTION,
      TRANSITION_NOT_AVAILABLE,
      UNKNOWN,
      ABORTED
};
```

- `EMPTY_DISC` – an empty disc is loaded
- `NO_DISC` – no disc is loaded
- `SHORT_CAPACITY` – the remaining capacity of the medium is smaller than the specified recording time. In this case, nothing has been recorded.
- `READ_ERR` – the disc cannot be read (e.g., because of dust or scratch)
- `WRITE_ERR` – the disc cannot be written (e.g., because of dust or scratch)
- `WRITE_PROTECTED` – the write-protected disc is loaded
- `DISC_FULL` – the loaded disc is fully recorded. (No recordable capacity remains.)
- `TRACK_FULL` – no recordable track remains. (The recordable capacity is available, but the track numbers are fully used.)
- `TOC_EDIT_BUSY` – the AV Disc is editing the disc's TOC just now.
- `LOCKED` – the AV Disc is "hold locked" (Some portable mobile devices have a small mechanical toggle switch called a "hold lock switch". While this switch is ON, i.e., the AV Disc is hold locked, the device is guarded against operations such as accidental power on when not in use, or interruption of play or record from accidental pressing of a front panel button or a GUI button.)
- `NO_CONNECTION` – no connection exists from the FCM plugs
- `TRANSITION_NOT_AVAILABLE` – vendor dependent functionality that inhibits the control command (e.g., the transition to or from a `RECORD` mode from any transport state except `STOP`)
- `UNKNOWN` – an unknown condition exists
- `ABORTED` – the operation invoked is not completed as intended because some other operation is requested from front panel button, IR remote, or other application, etc.

Direction

The AV Disc APIs use the Stream Manager `Direction` data type, see section 5.9.2.

6.6.3 AV Disc Terminology

track	an item appearing in the list obtained via `AvDisc::GetItemList`
first track	the item that has the lowest index which is not equal to zero
last track	the item that has the highest index which is not equal to zero
current track	the item that contains the current position

current position	In case the transport mechanism is stopped it identifies the beginning of the first track. In case the transport mechanism is rotating it identifies the playback or recording position on the medium corresponding to a specific FCM input or output plug.

6.6.4 AV Disc API

AvDisc::GetItemList

Prototype

```
Status AvDisc::GetItemList(
      in ushort listNumber,
      out sequence<ItemIndex> itemIndexList)
```

Parameters

- `ushort list` – specifies the list number of list to be returned.
- `itemIndexList` – a list of items and indexes. The safe parameter size limit is 512 `ItemIndex` values. Maximum size of HAVi message representing this API is 64 Kbytes

Description

This API returns a list containing information about contents in the loaded medium on the Disc device. The structure of lists is hierarchical. All lists derive from the root list with list number zero, and an item in a list may refer to a child list. This API is optional. Its support can be verified by the `AvDisc::GetCapability` API.

In the root list, the `ItemIndex` with index zero contains general disc information (disc title, total playback time of the disc, ...), while all other index values refer to individual lists. In the lists except the root list, the item index with index zero contains the list information (list title, total playback time of the list, …), while all other index value refer to individual tracks (track title, track playback time, ...) or to a child list. The `ItemIndex` that refers to a child list in a parent list is identical to the `ItemIndex` with index zero in the child list.

Error codes

- `ENOT_IMPLEMENTED` – the target device does not hold the `itemIndex` list internally.
- `AvDisc::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `AvDisc::GetRejectInfo`.

AvDisc::Play

Prototype

```
Status AvDisc::Play(
      in AvDiscPlayMode mode, in ushort plugNum,
      in ushort listNumber,
      in ushort indexNumber)
```

Parameter

- `mode` – specifies the playback mode
- `plugNum` – plug number of the source FCM plug used for playback
- `listNumber`– specifies the list number of the list of items.
- `indexNumber` – specifies the index number of the desired track. This parameter need not be specified (i.e., the value can be 0) when `mode` is `NORMAL`, `REPEAT_ALL`, `SHUFFLE` or `RANDOM`.

When mode is DIRECT_1, DIRECT or REPEAT_1 it specifies the starting track.

Description

This API plays the data on the loaded medium using the specified playback mode. Depending upon the value of mode, the following is performed:

- NORMAL – play from the current position. This mode is restricted to un-pause operation.
- DIRECT_1 – play the track specified by indexNumber, then stop.
- DIRECT – play tracks once from the track specified by indexNumber until the last track in order of increasing index.
- REPEAT_1 – play the track specified by indexNumber and repeat the track continuously.
- REPEAT_ALL – play all tracks in order of increasing index from the first to the last track. Repeat this action continuously.
- SHUFFLE – play each track once in random order, then stop.
- RANDOM – play each track in random order, continue playing indefinitely.

Error codes

- AvDisc::EREJECTED – the API could not be executed. Reject conditions may be retrieved via AvDisc::GetRejectInfo.
- EINVALID_PARAMETER – plugNum or indexNumber are not valid.
- AvDisc::ENOT_SUPPORTED – the target device does not support the specified playback mode.

AvDisc::Record

Prototype

```
Status AvDisc::Record(
      in AvDiscRecordingMode mode, in ushort plugNum,
      in ushort listNumber,
      in ushort indexNumber,
      in TimeCode recordingTime, in uint64 recordingSize)
```

Parameter

- mode – specifies the recording mode
- plugNum – plug number of the sink FCM plug used for recording
- listNumber– specifies the list number of desired track.
- indexNumber – specifies the index number of desired track
- recordingTime – specifies the total time of the content to be recorded. This parameter may be zero if it need not be specified.
- recordingSize – specifies the size (in bytes) of the content to be recorded. This parameter may be zero if it need not be specified.

Description

This API records contents in a track specified by indexNumber using the specified recording mode. This API is optional. Its support can be verified by the AvDisc::GetCapability API. Depending upon the value of mode, the following is performed:

- NORMAL – record from the current position. In this mode, indexNumber need not be specified (i.e., the value can be 0). No new track is created. This mode is restricted to the un-pause operation of the AvDisc::RecPause API, and recording in previous NEW, or OVERWRITE, or OVERWRITE_AND_JUMP mode.
- NEW – start a new track recording in an unused track. The new track becomes the last track with an index that equals the index of the previous last track plus one or it becomes the first track for

an empty disc.
- OVERWRITE – start recording a new track at the beginning of the track specified by indexNumber, or at the current position in case indexNumber is not specified (value 0). Continue recording if the current position exceeds the track specified by indexNumber, or the current track in case indexNumber is not specified.
- OVERWRITE_AND_JUMP – start recording a new track at the beginning of the track specified by indexNumber, or at the current position in case indexNumber is not specified (value 0). If the current position reaches the end of the specified track or the current track in case indexNumber is not specified, continue recording in unused space on the medium.

Error codes
- ENOT_IMPLEMENTED – the target device does not implement this API.
- AvDisc::EREJECTED – the API could not be executed. Reject conditions may be retrieved via AvDisc::GetRejectInfo.
- EINVALID_PARAMETER – mode, plugNum or indexNumber are not valid.
- AvDisc::ENOT_SUPPORTED – the target device does not support the specified recording mode.

AvDisc::VariableForward

Prototype
```
Status AvDisc::VariableForward(
      in ForwardSpeed speed, in ushort plugNum)
```

Parameter
- speed – forward speed mode
- plugNum – plug number of the source FCM plug used for playback

Description

This API plays the medium back at variable-speeds according to speed. The actual speeds encoded by the speed have the following restrictions. If the target device does not support the specified FAST_FORWARD_x (or SLOW_FORWARD_x) speed, then the device shall interpret it as FASTEST_FORWARD (or SLOWEST_FORWARD).

This variable-forward action is continued until the next API such as AvDisc::Play or AvDisc::Stop is issued or the end of the medium.

This API is optional. Its support can be verified by the AvDisc::GetCapability API.

Error codes
- ENOT_IMPLEMENTED – the target device does not implement this API.
- AvDisc::EREJECTED – the API could not be executed. Reject conditions may be retrieved via AvDisc::GetRejectInfo.
- EINVALID_PARAMETER – plugNum or speed are not valid.

AvDisc::VariableReverse

Prototype
```
Status AvDisc::VariableReverse(
      in ReverseSpeed speed, in ushort plugNum)
```

Parameter

- `reverseSpeed` – reverse speed mode
- `plugNum` – plug number of the source FCM plug used for playback

Description

This API plays the medium back at variable-speeds, in reverse, according to `speed`. The actual speeds encoded by the `speed` have the following restrictions. If the target device does not support the specified `FAST_REVERSE_x` (or `SLOW_REVERSE_x`) speed, then the device shall interpret it as `FASTEST_REVERSE` (or `SLOWEST_REVERSE`).

This variable-reverse action is continued until the next API such as `AvDisc::Play` or `AvDisc::Stop` is issued or the beginning of the medium.

This API is optional. Its support can be verified by the `AvDisc::GetCapability` API.

Error codes

- `ENOT_IMPLEMENTED` – the target device does not implement this API.
- `AvDisc::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `AvDisc::GetRejectInfo`.
- `EINVALID_PARAMETER` – `plugNum` or `speed` are not valid.

AvDisc::Stop

Prototype

```
Status AvDisc::Stop(
      in Direction dir, in ushort plugNum)
```

Parameter

- `dir` – direction of the FCM plug used for playback (or recording), indicates either a source (`OUT`) or sink (`IN`) plug
- `plugNum` – plug number of the FCM plug used for playback (or recording)

Description

This API stops all transport mechanism motion.

Error codes

- `AvDisc::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `AvDisc::GetRejectInfo`.
- `EINVALID_PARAMETER` – `plugNum` is not valid.

AvDisc::RecPause

Prototype

```
Status AvDisc::RecPause(in ushort plugNum)
```

Parameter

- `plugNum` – plug number of the sink FCM plug used for recording

Description

This API pauses the recording operation. This pause action is stopped when the next API such as

AvDisc::Record or AvDisc::Stop is issued.

This API is optional. Its support can be verified by the AvDisc::GetCapability API. If the AvDisc::Record API is supported this API should also be supported.

Error codes

- ENOT_IMPLEMENTED – the target device does not implement this API.
- AvDisc::EREJECTED – the API could not be executed. Reject conditions may be retrieved via AvDisc::GetRejectInfo.
- EINVALID_PARAMETER – plugNum is not valid.

AvDisc::Skip

Prototype

```
Status AvDisc::Skip(
      in SkipDirection direction, in SkipMode mode,
      in long count, in ushort plugNum)
```

Parameters

- direction – skip direction
- mode – skip mode
- count – number of items to be skipped, in accordance with the skipMode parameter. If the skip mode is RELATIVE_TIME or ABSOLUTE_TIME, the format of this parameter is 0xHHMMSSFF where HH specifies hour, MM specifies minute, SS specifies second, and FF specifies frame, each in one byte.
- plugNum – plug number of the source FCM plug used for playback.

Description

This API fast-forwards (or rewinds) the medium in the skip-mode specified by the mode parameter, skipping the number of items (in accordance with the mode parameter) specified by the count parameter, then playbacks.

The SUCCESS response for this API should be returned after completion of skip operation. If the skip operation is aborted before its completion because of user's operation through the front panel button, IR remote or the other application, this API returns AvDisc::EREJECTED and set the reject condition to ABORTED.

This API is optional. Its support can be verified by the AvDisc::GetCapability API.

Error codes

- ENOT_IMPLEMENTED – the target device does not implement this API.
- EINVALID_PARAMETER – direction, mode, count or plugNum are not valid.
- AvDisc::EREJECTED – the API could not be executed. Reject conditions may be retrieved via AvDisc::GetRejectInfo.
- AvDisc::ENOT_SUPPORTED – the target device does not support the specified skip mode.

AvDisc::InsertMedia

Prototype

```
Status AvDisc::InsertMedia()
```

Description

This API inserts the medium into the Disc.

Error codes

- `AvDisc::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `AvDisc::GetRejectInfo`.

AvDisc::EjectMedia

Prototype

```
Status AvDisc::EjectMedia()
```

Description

This API ejects the currently loaded medium from the target device.

Error codes

- `AvDisc::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `AvDisc::GetRejectInfo`.

AvDisc::GetState

Prototype

```
Status AvDisc::GetState(
      out AvDiscTransportState state,
      in Direction dir, in ushort plugNum)
```

Parameters

- `state` – status of the target device's transport mechanism
- `dir` – type of the plug to be examined, either a source (`OUT`) or sink (`IN`) FCM plug
- `plugNum` – plug number of the FCM plug desired to be examined

Description

This API returns the current status of the target device's transport mechanism (playing, recording, etc.) on the specified plug.

Error codes

- `EINVALID_PARAMETER` – `plugNum` is not valid.

AvDisc::GetFormat

Prototype

```
Status AvDisc::GetFormat(
      out MediaFormatId format,
      out WriteProtectStatus writeStatus)
```

Parameters

- `format` – type of the medium loaded in the target device. See Annex 11.10 for possible values. *Note* – `format` is not relevant to whether there is isochronous data in the target device or not.

- writeStatus – whether it is possible to write on the medium loaded in the target device

Description

This API returns information concerning the medium loaded in the target device (medium type, whether the medium can be recorded to, etc.)

Error codes

- AvDisc::EREJECTED – the API could not be executed. Reject conditions may be retrieved via AvDisc::GetRejectInfo.

AvDisc::GetPosition

Prototype

```
Status AvDisc::GetPosition(
      in AvDiscCounterType type,
      in Direction dir,
      in ushort plugNum,
      out AvDiscCounterValue value)
```

Parameters

- type – type of description used for indicating the current position
- dir – type of the plug to be examined, either a source (OUT) or sink (IN) FCM plug.
- plugNum – plug number of the FCM plug associated with stream desired to be examined.
- value – returned value which indicates the current position of the target device in the description specified by type

Description

This API gets the current position of the target device in the format specified by the type parameter for the specified plug. The counter type TRACK_NUMBER is mandatory and RELATIVE_TIME and ABSOLUTE_TIME are optional. If type is either RELATIVE_TIME or ABSOLUTE_TIME and the counter type is not supported then AvDisc::GetPosition returns AVDISC::ENOT_SUPPORTED.

Error codes

- AvDisc::EREJECTED – the API could not be executed. Reject conditions may be retrieved via AvDisc::GetRejectInfo.
- AvDisc::ENOT_SUPPORTED – the requested type is not supported.
- EINVALID_PARAMETER – type or plugNum is not valid.

AvDisc::Erase

Prototype

```
Status AvDisc::Erase(
      In ushort listNumber,
      in ushort indexNumber)
```

Parameters

- listNumber – specifies the list number of desired track.
- indexNumber – specifies the index number of the contents desired to be erased. The value 0 indicates that all contents in the list are to be erased. If both listNumber and indexNumber are 0, it indicates that all contents in the disc are to be erased.

Description

This API erases the contents specified by `indexNumber` (or all contents).

Error codes

- `ENOT_IMPLEMENTED` – the target device does not implement this API.
- `AvDisc::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `AvDisc::GetRejectInfo`.
- `EINVALID_PARAMETER` – `indexNumber` is not valid.

AvDisc::PutItemList

Prototype

```
Status AvDisc::PutItemList(
      In ushort listNumber,
      in sequence<ItemIndex> itemIndexList)
```

Parameters

- `listNumber` – specifies the list number of list to be written.
- `itemIndexList` – a list of items and indexes. The definitions of the `ItemIndex` structure are described above. The safe parameter size limit is 512 `ItemIndex` values. Maximum size of HAVi message representing this API is 64 Kbytes

Description

This API changes the specified `itemIndexList` and makes the target device write the corresponding information on the disc media.

Error codes

- `ENOT_IMPLEMENTED` – the target device does not implement this API.
- `AvDisc::EREJECTED` – the API could not be executed. Reject conditions may be retrieved via `AvDisc::GetRejectInfo`.

AvDisc::GetCapability

Prototype

```
Status AvDisc::GetCapability(
      out sequence<boolean> capabilityList,
      out sequence<MediaFormatId> playFormats,
      out sequence<MediaFormatId> recordFormats)
```

Parameters

- `capabilityList` – a list of capabilities supported by the AV Disc Functional Component Module. The i^{th} member of `capabilityList` indicates the availability of the capability identified by `AvDiscCapability` value *i*. The safe parameter size is 32 `boolean` values.
- `playFormats` – the disc formats the AV Disc is capable of playing. The safe parameter size limit is 14 `MediaFormatId` values.
- `recordFormats` – the disc formats the AV Disc is capable of recording. The safe parameter size limit is 14 `MediaFormatId` values.

Description

This API returns the capabilities of the AV Disc Functional Component Module.

AvDisc::GetRejectInfo

Prototype

```
Status AvDisc::GetRejectInfo (
   out sequence<AvDiscRejectCondition> rejectedConditions,
   out OperationCode rejectedOpcode,
   out sequence<AvDiscRejectCondition> currentRejectConditions)
```

Parameters

- `rejectedConditions` – contains one or more conditions for which the latest `AvDisc::EREJECTED` error occurred. It is independent of whether the conditions causing the error still exist or have disappeared. Only if no `AvDisc::EREJECTED` error has occurred will the list will be empty. The safe parameter size limit is *n* `AvDiscRejectCondition` values, where *n* is the number of members of `AvDiscRejectCondition`.
- `rejectedOpcode` – the operation code of the last invoked API that caused a `AvDisc::EREJECTED` status. Undefined in case `rejectedConditions` is empty.
- `currentRejectConditions` – contains one or more conditions that currently exist that would cause one or more APIs to return `AvDisc::EREJECTED`. If no reject conditions currently exist the list will be empty. The safe parameter size limit is *n* `AvDiscRejectCondition` values, where *n* is the number of members of `AvDiscRejectCondition`.

The possible reject conditions for each AV Disc API service are listed below.

Service	Possible Conditions
`AvDisc::GetItemList`	`EMPTY_DISC, NO_DISC, READ_ERR, TOC_EDIT_BUSY, LOCKED`
`AvDisc::Play`	`EMPTY_DISC, NO_DISC, READ_ERR, TOC_EDIT_BUSY, LOCKED, TRANSITION_NOT_AVAILABLE`
`AvDisc::Record`	`SHORT_CAPACITY, NO_DISC, WRITE_ERR, WRITE_PROTECTED, DISC_FULL, TRACK_FULL, TOC_EDIT_BUSY, LOCKED, NO_CONNECTION, TRANSITION_NOT_AVAILABLE`
`AvDisc::VariableForward`	`EMPTY_DISC, NO_DISC, READ_ERR, TOC_EDIT_BUSY, LOCKED, TRANSITION_NOT_AVAILABLE`
`AvDisc::VariableReverse`	`EMPTY_DISC, NO_DISC, READ_ERR, TOC_EDIT_BUSY, LOCKED, TRANSITION_NOT_AVAILABLE`
`AvDisc::Stop`	`NO_DISC, LOCKED`
`AvDisc::RecPause`	`NO_DISC, WRITE_PROTECTED, DISC_FULL, TRACK_FULL, TOC_EDIT_BUSY, LOCKED, NO_CONNECTION, TRANSITION_NOT_AVAILABLE`
`AvDisc::Skip`	`EMPTY_DISC, NO_DISC, READ_ERR, TOC_EDIT_BUSY, LOCKED,TRANSITION_NOT_AVAILABLE, ABORTED`
`AvDisc::InsertMedia`	`LOCKED, TRANSITION_NOT_AVAILABLE`

AvDisc::EjectMedia	NO_DISC, TOC_EDIT_BUSY, LOCKED, TRANSITION_NOT_AVAILABLE
AvDisc::GetFormat	LOCKED
AvDisc::GetPosition	NO_DISC, LOCKED
AvDisc::Erase	EMPTY_DISC, NO_DISC, READ_ERR, WRITE_ERR, WRITE_PROTECTED, TOC_EDIT_BUSY, LOCKED, TRANSITION_NOT_AVAILABLE
AvDisc::PutItemList	EMPTY_DISC, NO_DISC, WRITE_ERR, WRITE_PROTECTED, DISC_FULL, TOC_EDIT_BUSY, LOCKED, TRANSITION_NOT_AVAILABLE

Description

This API returns information on the conditions that caused or could cause `AvDisc::EREJECTED` to be returned as a result of the AV Disc APIs.

6.6.5 AV Disc Events

AvDiscItemListChanged

Prototype

```
void AvDiscItemListChanged()
```

Description

This API notifies changes of contents in the `ItemIndex` list the AV Disc holds internally. Applications which use the `ItemIndex` list should retrieve it again.

AvDiscStateChanged

Prototype

```
void AvDiscStateChanged(
      in AvDiscTransportState state,
      in Direction dir,
      in ushort plugNum)
```

Parameters

- `state` – see the `AvDisc::GetState` API
- `dir` – type of the plug which has changed its state, i.e., either a source (`OUT`) or sink (`IN`) plug
- `plugNum` – plug number of the FCM plug which changed its state

Description

This API notifies changes in the value of the current state as defined in the `GetState` API.

6.6.6 AV Disc Notification Attributes

AvDisc::currentState

Attribute

```
struct AvDiscCurrentState {
     AvDiscTransportState  state;
     Direction             dir;
     ushort                plugNum;
} currentState
```

Description

New setting of `AvDiscTransportState`. The plug involved is also indicated.

6.7 Amplifier FCM

This FCM supports basic functions of an audio amplifier device. It does not support advanced functions such as surround sound. Basic function of Amplifier is to provide audio interface to users, so it is recommended clients not to reserve Amplifier FCMs without specific purpose or reason, according to recommendation described in 3.8.1.

6.7.1 Amplifier Services

Service	Comm Type	Locality	Access	Resv Prot
Amplifier::SetVolume	M	global	all	yes
Amplifier::GetVolume	M	global	all	
Amplifier::SetMute	M	global	all	yes
Amplifier::GetMute	M	global	all	
Amplifier::SetBalance	M	global	all	yes
Amplifier::GetBalance	M	global	all	
Amplifier::SetLoudness	M	global	all	yes
Amplifier::GetLoudness	M	global	all	
Amplifier::GetCapability	M	global	all	
Amplifier::SetEqualizer	M	global	all	yes
Amplifier::GetEqualizer	M	global	all	
Amplifier::GetEqualizerCapability	M	global	all	
Amplifier::SetPresetMode	M	global	all	yes
Amplifier::GetPresetMode	M	global	all	
Amplifier::GetPresetCapability	M	global	all	
Amplifier::GetAudioLatency	M	global	all	

6.7.2 Amplifier Data Structures

AmplifierCapability

Definition

```
enum AmplifierCapability {
      BALANCE,
      LOUDNESS
};
```

Description

This defines a set of basic amplifier capabilities.

EqualizerFrequency

Definition

```
struct EqualizerFrequency {
      short lowestFrequency;
      short highestFrequency;
}
```

Description

This structure represents the lowest frequency (Hz) and the highest frequency (Hz) of one equalizer band.

AmplifierPresetMode

Definition

```
enum AmplifierPresetMode {
      OFF, SPEECH, MOVIE,
      MUSIC_CLASSICAL, MUSIC_JAZZ, MUSIC_ROCK
};
```

Description

Each value represents a specific audio mode. The settings associated with each mode are determined by the manufacturer.

Element	Description
OFF	Indicates that no preset is active.
SPEECH	Indicates the preferred settings for reproducing human speech.
MOVIE	Indicates the preferred settings for reproducing a movie sound track. This typically is good for a combination of speech, music and sound effects.
MUSIC_CLASSICAL	Indicates the preferred settings for reproducing classical music. This typically produces a clean, natural sound.
MUSIC_JAZZ	Indicates the preferred settings for reproducing jazz music. This typically produces a bright sound.
MUSIC_ROCK	Indicates the preferred settings for reproducing rock or pop music. This typically produces a powerful sound.

6.7.3 Amplifier API

Amplifier::SetVolume

Prototype

```
Status Amplifier::SetVolume(
      in octet volumeValue)
```

Parameters

- `volumeValue` – volume value of the amplifier device to be set
  0 : minimum volume
  255 : maximum volume

Description

This API sets the volume.

Amplifier::GetVolume

Prototype

```
Status Amplifier::GetVolume(
      out octet volumeValue)
```

Parameters

- `volumeValue` – current volume value

Description

This API gets the current volume value of the amplifier device.

Amplifier::SetMute

Prototype

```
Status Amplifier::SetMute(
      in boolean muteState)
```

Parameters

- `muteState` – mute state of amplifier device to be set
  `True` : mute on
  `False` : mute off

Description

This API sets the mute state of the amplifier device.

Amplifier::GetMute

Prototype

```
Status Amplifier::GetMute(
      out boolean muteState)
```

Parameters

- `muteState` – Current mute state

Description

This API gets the current mute state of amplifier device.

Amplifier::SetBalance

Prototype

```
Status Amplifier::SetBalance(
      in octet balanceValue)
```

Parameters

- `balanceValue` – LR balance value of amplifier device to be set
  0 : L only (L output is saturated, R output is silent)
  127 : center (both L and R outputs are saturated)
  254 : R only (R output is saturated, L output is silent)

Description

This API sets the LR balance value of the amplifier device.

Error codes

- `ENOT_IMPLEMENTED` – the amplifier does not support adjustable balance
- `EINVALID_PARAMETER` – the balance setting is invalid (= 255)

Amplifier::GetBalance

Prototype

```
Status Amplifier::GetBalance(
      out octet balanceValue)
```

Parameters

- `balanceValue` – current LR balance value

Description

This API gets current the LR balance value of the amplifier device.

Error codes

- `ENOT_IMPLEMENTED` – the amplifier does not support adjustable balance

Amplifier::SetLoudness

Prototype

```
Status Amplifier::SetLoudness(
      in boolean loudnessState)
```

Parameters

- `loudnessState` – loudness state to be set

Description

This API sets the loudness state of the amplifier device.

Error codes

- `ENOT_IMPLEMENTED` – the amplifier does not support loudness mode

Amplifier::GetLoudness

Prototype

```
Status Amplifier::GetLoudness(
      out boolean loudnessState)
```

Parameters

- `loudnessState` – current loudness state

Description

This API gets the current loudness state of the amplifier device.

Error codes

- `ENOT_IMPLEMENTED` – the amplifier does not support loudness mode

Amplifier::GetCapability

Prototype

```
Status Amplifier::GetCapability(
      out sequence<boolean> capabilityList)
```

Parameters

- `capabilityList` – list of amplifier capability types. The i^{th} member of `capabilityList` indicates the availability of the capability identified by `AmplifierCapability` value *i*. The safe parameter size is 32 `boolean` values.

Description

Returns a list of amplifier capabilities.

Amplifier::SetEqualizer

Prototype

```
Status Amplifier::SetEqualizer(
      in sequence<octet> equalizerValue)
```

Parameters

- `equalizerValue` – equalizer values to be set. In this array of bytes, the first byte defines the equalizer value for the lowest frequency band and the last byte defines the equalizer value for the highest frequency band.
  0 : minimum output value
  127 : center (output value as in original input stream)
  254 : maximum output value.

The length of the `equalizerValue` list should be identical to the length of the `EqualizerFrequency` list that can be obtained via `GetEqualizerCapability` (the error when this is not the case is `EINVALID_PARAMETER`).

Description

This API sets the intensity of the equalizer bands.

Error codes

- `ENOT_IMPLEMENTED` – the amplifier does not support adjustable equalization
- `EINVALID_PARAMETER` – one or more equalizer settings are invalid (= 255)

Amplifier::GetEqualizer

Prototype

```
Status Amplifier::GetEqualizer(
```

```
      out sequence<octet> equalizerValue)
```

Parameters

- `equalizerValue` – current equalizer values for the amplifier device. In this array of bytes, the first byte defines the equalizer value for the lowest frequency band and the last byte defines the equalizer value for the highest frequency band.

Description

This API gets the intensity list of the equalizer bands.

Error codes

- `ENOT_IMPLEMENTED` – the amplifier does not support adjustable equalization

Amplifier::GetEqualizerCapability

Prototype

```
Status Amplifier::GetEqualizerCapability(
      out sequence<EqualizerFrequency> list)
```

Parameters

- `list` – The list of bandwidth information that the equalizer can support. The safe parameter size limit is 20 `EqualizerFrequency` values.

Description

This API returns the list of bandwidth information that the equalizer can support. When an amplifier device has no equalizer functionality, this API returns an empty `EqualizerFrequency` list (length is zero).

Amplifier::SetPresetMode

Prototype

```
Status Amplifier::SetPresetMode(
      in  AmplifierPresetMode amplifierPresetMode)
```

Parameters

- `amplifierPresetMode` – preset mode to be set

Description

This API sets the preset mode of the amplifier device.

Error codes

- `ENOT_IMPLEMENTED` – the amplifier does not support preset modes

Amplifier::GetPresetMode

Prototype

```
Status Amplifier::GetPresetMode(
      out AmplifierPresetMode amplifierPresetMode)
```

Parameters

- `amplifierPresetMode` – current preset mode

Description

This API gets the current preset mode of the amplifier device.

Error codes

- `ENOT_IMPLEMENTED` – the amplifier does not support preset modes

Amplifier::GetPresetCapability

Prototype

```
Status Amplifier::GetPresetCapability(
      out sequence<boolean> capabilityList)
```

Parameters

- `capabilityList` – capability list of preset modes. The i^{th} member of `capabilityList` indicates the availability of the capability identified by `AmplifierPresetMode` value *i*. The safe parameter size is 32 `boolean` values.

Description

Returns a list of supported preset modes.

Amplifier::GetAudioLatency

Prototype

```
Status Amplifier::GetAudioLatency(out short latency)
```

Parameters

- `latency` – the number of milliseconds between the amplifier input and output.

Description

This API allows the nominal latency of the amplifier to be measured. This is useful in cases where audio must be synchronized – especially when signal processing algorithms add long delays.

6.7.4 Amplifier Notification Attributes

Amplifier::volume

Attribute

```
octet volume
```

Description

New volume setting.

Amplifier::mute

Attribute

```
boolean mute
```

Description

New mute setting.

Amplifier::balance

Attribute

```
octet balance
```

Description

New balance setting.

Amplifier::loudness

Attribute

```
boolean loudness
```

Description

New loudness setting.

Amplifier::equalizer

Attribute

```
sequence<octet> equalizer
```

Description

New equalizer settings (only useful with comparator `ANY`).

6.8 Display FCM

The Display API supports the following functions:

- Filtering function for displaying the video.
- Set up of screen and window modes.
- Assignment of an input FCM plug to a logical window.

Basic function of displays is to provide visual interface to users, so it is recommended clients not to reserve Display FCMs without specific purpose or reason, according to recommendation described in 3.8.1.

6.8.1 Display Services

Service	Comm Type	Locality	Access	Resv Prot
Display::SetContrast	M	global	all	yes
Display::GetContrast	M	global	all	
Display::SetTint	M	global	all	yes
Display::GetTint	M	global	all	
Display::SetColor	M	global	all	yes
Display::GetColor	M	global	all	
Display::SetBrightness	M	global	all	yes
Display::GetBrightness	M	global	all	
Display::SetSharpness	M	global	all	yes
Display::GetSharpness	M	global	all	
Display::GetCapability	M	global	all	
Display::GetStandardPictureValue	M	global	all	
Display::SetPresetMode	M	global	all	yes
Display::GetPresetMode	M	global	all	
Display::GetPresetCapability	M	global	all	
Display::SetScreenMode	M	global	all	yes
Display::GetScreenMode	M	global	all	
Display::SetWindowMode	M	global	all	yes
Display::GetWindowMode	M	global	all	
Display::SetActiveWindow	M	global	all	yes
Display::GetActiveWindow	M	global	all	
Display::GetWindowRectangle	M	global	all	
Display::AssignPlugToDisplay	M	global	all	yes
Display::GetVideoLatency	M	global	all	

6.8.2 Display Data Structures

DisplayCapability

```
enum DisplayCapability {
      PICTURE_CONTRAST ,
      PICTURE_TINT ,
      PICTURE_COLOR ,
      PICTURE_BRIGHTNESS ,
      PICTURE_SHARPNESS ,
      SCREEN_NORMAL ,
      SCREEN_WIDE ,
      WINDOW_SINGLE ,
      WINDOW_DOUBLE ,
      WINDOW_PinP
};
```

PictureAttribute

```
enum PictureAttribute {
      CONTRAST,
      TINT,
      COLOR,
      BRIGHTNESS,
      SHARPNESS
};
```

ScreenMode

```
enum ScreenMode {NORMAL, WIDE};
```

Description

Possible screen modes:

`NORMAL`	4:3 aspect ratio
`WIDE`	16:9 aspect ratio

WindowMode

```
enum WindowMode {SINGLE, DOUBLE, PinP};
```

Description

Possible window modes:

`SINGLE`	single window mode
`DOUBLE`	double window mode
`PinP`	PinP mode

DisplayPresetMode

Definition

```
enum DisplayPresetMode {
      NORMAL,
      USER_PRESET,
      MOVIE,
      GAME
};
```

Description

Each value represents a specific genre. The settings associated with each mode are determined by the manufacturer.

Element	Description
NORMAL	Indicates the default settings for the display.
USER_PRESET	Refers to settings which were configured by the user in a proprietary way.
MOVIE	Indicates the preferred settings for viewing a movie, and presenting the “film” look.
GAME	Indicates the preferred settings for viewing video games and computer graphics.

6.8.3 Display API

Display::SetContrast

Prototype

```
Status Display::SetContrast(
      in octet contrastValue)
```

Parameters

- `contrastValue` – contrast value to be set
  - 0 : minimum contrast value
  - 127 : center
  - 255 : maximum contrast value

Description

This API sets the contrast value of the display device. Contrast is picture contrast.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::GetContrast

Prototype

```
Status Display::GetContrast(
      out octet contrastValue)
```

Parameters

- `contrastValue` – current contrast value

Description

This API gets the contrast value of the display device.

Error codes

- `ENOT_IMPLEMENTED`

Display::SetTint

Prototype

```
Status Display::SetTint(
      in octet tintValue)
```

Parameters

- `tintValue` – tint value to be set
  0 : make skin tones purplish
  127 : center
  255 : make skin tones greenish

Description

This API sets the tint value of the display device.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::GetTint

Prototype

```
Status Display::GetTint(
      out octet tintValue)
```

Parameters

- `tintValue` – current value of tint

Description

This API gets the current tint value of display device.

Error codes

- `ENOT_IMPLEMENTED`

Display::SetColor

Prototype

```
Status Display::SetColor(
      in octet colorValue)
```

Parameters

- `colorValue` – color value to be set
  0 : minimum color intensity
  127 : center
  255 : maximum color intensity

Description

This API sets the color value of the display device. Color is color intensity.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::GetColor

Prototype

```
Status Display::GetColor(
      out octet colorValue)
```

Parameters

- `colorValue` – current color value

Description

This API gets the current color value of the display device.

Error codes

- `ENOT_IMPLEMENTED`

Display::SetBrightness

Prototype

```
Status Display::SetBrightness(
      in octet brightnessValue)
```

Parameters

- `brightnessValue` – brightness value to be set
  0 : minimum brightness value
  127 : center
  255 : maximum brightness value

Description

This API sets the brightness value of the display device. Brightness is a level of blackness.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::GetBrightness

Prototype

```
Status Display::GetBrightness (
```

```
      out octet brightnessValue)
```

Parameters

- `brightnessValue` – current brightness value

Description

This API gets the current brightness value of the display device.

Error codes

- `ENOT_IMPLEMENTED`

Display::SetSharpness

Prototype

```
Status Display::SetSharpness(
      in octet sharpnessValue)
```

Parameters

- `sharpnessValue` – sharpness value to be set
  - 0 : minimum sharpness value
  - 127 : center
  - 255 : maximum sharpness value

Description

This API sets the sharpness value of the display device. Sharpness is sharpness of outline.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::GetSharpness

Prototype

```
Status Display::GetSharpness(
      out octet sharpnessValue)
```

Parameters

- `sharpnessValue` – current sharpness value

Description

This API gets the current sharpness value of the display device.

Error codes

- `ENOT_IMPLEMENTED`

Display::GetCapability

Prototype

```
Status Display::GetCapability(
      out sequence<boolean> capabilityList)
```

Parameters

- `capabilityList` – list of display capabilities. The i^{th} member of `capabilityList` indicates the availability of the capability identified by `DisplayCapability` value *i*. The safe parameter size is 32 `boolean` values.

Description

Provides a list of capabilities supported by the display device.

Display::GetStandardPictureValue

Prototype

```
Status Display::GetStandardPictureValue(
      in PictureAttribute type,
      out octet standardValue)
```

Parameters

- `type` – the attribute for which the standard value is to be obtained
- `standardValue` – standard value of the specified picture attribute

Description

Returns a standard value for a specific picture attribute.

For attributes such as `CONTRAST` and `BRIGHTNESS` the standard value is not always 50% of the maximum. The standard value of a given attribute may differ for each manufacture and model.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::SetPresetMode

Prototype

```
Status Display::SetPresetMode(
      in  DisplayPresetMode displayPresetMode)
```

Parameters

- `displayPresetMode` – preset mode to be set

Description

This API sets the display device to a preset mode.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::GetPresetMode

Prototype

```
Status Display::GetPresetMode(
      out DisplayPresetMode displayPresetMode)
```

Parameters

- `displayPresetMode` – current preset mode

Description

This API gets the current preset mode of the display device.

Error codes

- `ENOT_IMPLEMENTED`

Display::GetPresetCapability

Prototype

```
Status Display::GetPresetCapability(
      out sequence<boolean> capabilityList)
```

Parameters

- `capabilityList` – capability list of preset modes. The i^{th} member of `capabilityList` indicates the availability of the capability identified by `DisplayPresetMode` value *i*. The safe parameter size is 32 `boolean` values.

Description

Returns a list of supported preset modes.

Display::SetScreenMode

Prototype

```
Status Display::SetScreenMode(
      in ScreenMode type)
```

Parameters

- `type` – screen mode type to be set

Description

This API sets the screen mode of the display device. Normal mode (4:3 aspect ratio) and wide mode (16:9 aspect ratio) are switched by setting the screen mode.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::GetScreenMode

Prototype

```
   Status Display::GetScreenMode(
           out ScreenMode type)
```

Parameters

- `type` – current screen mode

Description

This API gets the current screen mode of the display device.

Error codes
- ENOT_IMPLEMENTED

Display::SetWindowMode

Prototype
```
Status Display::SetWindowMode(
      in  WindowMode type)
```

Parameters
- type – window mode to be set

Description

This API supports multiple windows.

- In case of single window mode, the active window is displayed.
- In case of double window mode, the active window is displayed on the left side and the inactive window on the right side.
- In case of PinP mode, the active window is on the main screen and the inactive window is in the PinP window.

For swapping the active window and the inactive window see Display::SetActiveWindow.

Error codes
- ENOT_IMPLEMENTED
- EINVALID_PARAMETER

Display::GetWindowMode

Prototype
```
Status Display::GetWindowMode(
      out WindowMode type)
```

Parameters
- type – current window mode

Description

This API gets the current window mode of the display device.

Error codes
- ENOT_IMPLEMENTED

Display::SetActiveWindow

Prototype
```
Status Display::SetActiveWindow(
```

```
      in octet windowNum)
```

Parameters

- `windowNum` – active window number to be set

Description

This API sets the active window number.

A display device that supports double window mode and PinP mode has two logical windows. The windows are called window 0 (`windowNum` = 0) and window 1 (`windowNum` = 1).

In case of double window mode and PinP mode, locations for the two windows are changed by specifying the window number of the active window.

Error codes

- `ENOT_IMPLEMENTED`
- `EINVALID_PARAMETER`

Display::GetActiveWindow

Prototype

```
Status Display::GetActiveWindow(
      out octet windowNum)
```

Parameters

- `windowNum` – current active window number

Description

This API gets the current active window number of the display device.

Error codes

- `ENOT_IMPLEMENTED`

Display::GetWindowRectangle

Prototype

```
Status Display::GetWindowRectangle(
      in octet windowNum,
      out ushort xPosition,
      out ushort yPosition,
      out ushort width,
      out ushort height)
```

Parameters

- `windowNum` – number of the window
- `xPosition` – offset from the left edge of the display in pixels
- `yPosition` – offset from the top of the display in pixels
- `width` – width of the window in pixels
- `height` – height of the window in pixels

Description

This API returns the screen area of a window relative to the upper left corner of the display area. If the `WindowMode` is `SINGLE`, a query on the secondary window will return a zero size width and height.

Error codes

- `EINVALID_PARAMETER` – `windowNum` is not supported

Display::AssignPlugToDisplay

Prototype

```
Status Display::AssignPlugToDisplay(
      in ushort plugNum,
      in octet windowNum)
```

Parameters

- `plugNum` – input FCM plug number
- `windowNum` – window number which is assigned to the plug

Description

This API is used to apply a video substream from an input FCM plug to a specific logical window. If the plug has multiple video substreams and window 1 is specified the second video substream will be applied. In all other cases the first video substream will be applied to the window.

Error codes

- `EINVALID_PARAMETER`

Display::GetVideoLatency

Prototype

```
Status Display::GetVideoLatency(
      in octet windowNum,
      out short latency)
```

Parameters

- `windowNum` – the window for which the latency is to be measured
- `latency` – the number of milliseconds from the input FCM plug to visible display

Description

In the case of displays with decoders and other digital processing the latency can be high enough to cause a noticeable lag between the sound and the picture. This API allows the lag to be measured, so that it can be compensated elsewhere in the system.

Error codes

- `EINVALID_PARAMETER`

6.8.4 Display Notification Attributes

Display::contrast

Attribute
`octet contrast`

Description

New contrast setting.

Display::tint

Attribute
`octet tint`

Description

New tint setting.

Display::color

Attribute
`octet color`

Description

New color setting.

Display::brightness

Attribute
`octet brightness`

Description

New brightness setting.

Display::sharpness

Attribute
`octet sharpness`

Description

New sharpness setting.

Display::screenMode

Attribute

```
ScreenMode screenMode
```

Description

New screen mode setting.

Display::windowMode

Attribute

```
WindowMode windowMode
```

Description

New window mode setting.

Display::activeWindow

Attribute

```
octet activeWindow
```

Description

New active window setting.

Display::presetMode

Attribute

```
DisplayPresetMode presetMode
```

Description

New preset mode setting.

Display::windowRectangle

Attribute

```
struct WindowRectangle {
      octet windowNum;
      ushort xPosition;
      ushort yPosition;
      ushort width;
      ushort height;
} windowRectangle
```

- `windowNum` – number of the window
- `xPosition` – offset from the left edge of the display in pixels
- `yPosition` – offset from the top of the display in pixels
- `width` – width of the window in pixels
- `height` – height of the window in pixels

Description

New window size and position.

6.9 AV Display FCM

The AV Display FCM is a combination of the Display FCM and the Amplifier FCM. This is offered to simplify the pairing of the display and amplifier, as in a normal television set. It also allows the manufacturer to internally synchronize the audio and video, when these sources come from a single transport stream.

The definition for the AV Display FCM is obtained by combining the APIs for the Display FCM and the Amplifier FCM. Please refer to those sections for detailed definitions.

The only difference is that `Display::SetActiveWindow` selects the associated audio (sub) stream as well as the video (sub) stream.

6.10 Modem FCM

The aim of this chapter is to define a Modem FCM interface and specify the basic operations that a software element can use to establish asynchronous or isochronous connections over any outside network and to transfer any amount of data.

Using a modem within an HAVi network is reduced to calls of the Modem FCM APIs. A software element cannot know all features of every modem, thus should not need to know all commands for controlling modems. The Modem FCM internally supports the details of modem control in order to perform the communication required by the calling software element.

Typical applications:

- the Modem FCM can be used by all applications that need modem access: return channel, fax, BBS, data, data/voice, Internet... applications (it can also be used by the Web Proxy FCM).
- the Modem FCM can be used for isochronous connections, directly streaming data between modem plugs and FCM plugs associated with the client.
- another use would be to interface calls coming from the outside (e.g. the user, while on holiday location) and so receive remote commands intended for local HAVi devices.

6.10.1 Modem Protocol

6.10.1.1 Asynchronous Connections

This Modem FCM provides a software element with facilities to transfer data over a specified network through a modem by opening one or several asynchronous connections between itself and one Modem FCM. The number of connections that can be opened depends on the Modem FCM capabilities (it may be possible for a Modem FCM to handle several modems).

Opening such connections might have two different meanings, from the point of view of a client software element:

- the software element wants to initiate an outgoing call before transferring data (this is the *call mode*), or
- the software element wants to wait for an incoming call, and then transfer data after connection (this is the *answer mode*).

Once a connection is opened, and when the connection is established between the local modem and the remote modem, the software element sends to the Modem FCM the message to transfer, along with the needed transmission configuration (modem control for setup, off-hook and/or dialup, connection, and disconnection are transparent from the client side). When the Modem FCM receives data from the outside, it forwards the data to the client software element.

At the end of communication, the client software element has to close the connection between itself and the Modem FCM.

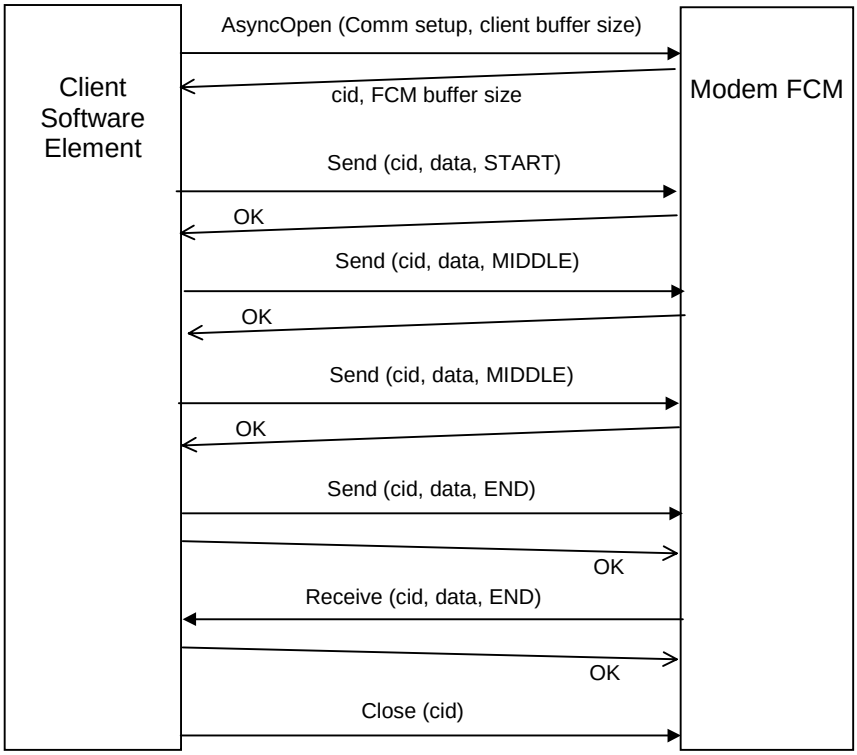

Figure 35. Asynchronous Modem FCM Communication

In Figure 35, a client software element opens a connection with the Modem FCM, specifying the communication setup (phone number to call, transfer rate...) and its buffer size (to segment message data if needed). If the required connection is possible, the Modem FCM returns an identifier for this communication, and its own buffer size.

The buffer size exchange between the client software element and the Modem FCM during Open phase allows the data to be sent (or received) to be split into several write actions: START indicates that the accompanying data is the first part of a multi-segment transfer, MIDDLE indicates that another data segment will come afterwards, and END indicates that the accompanying data are the last part of the data being transferred. Each message is sent after the response from the previous transfer is received. “OK” in the picture represents the acknowledgement of a transfer(see the description later in API section).

The client software element can close the connection when it wants.

6.10.1.2 Isochronous Connections

One or several isochronous connections can be established between a FCM associated with the client software element and the Modem FCM (according to FCM Modem capabilities). Before establishing these connections, the client software element must know which plugs (the plug numbers of its associated FCM and the plug numbers of the modem FCM) will be used to transfer isochronous data.

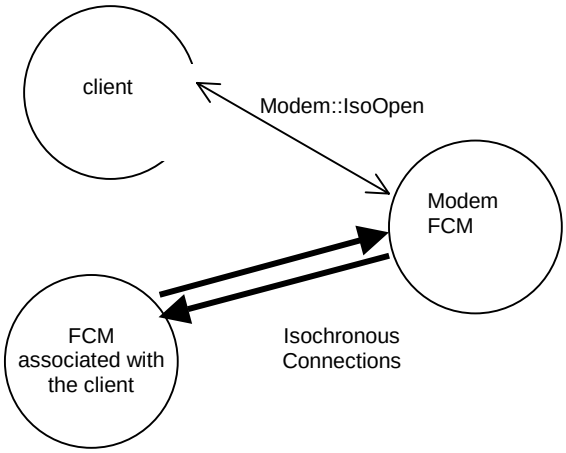

Figure 36. Isochronous Modem FCM Communication

Once a connection between these plugs is established (using the Stream Manager), the client software element can open a connection between itself and the Modem FCM. Note that in the case the client is a FCM, then the "FCM associated with the client" could be the client itself.

Supporting isochronous connections between the client software element and the Modem FCM is optional.

6.10.2 Modem Services

Service	Comm Type	Locality	Access	Resv Prot
Modem::AsyncOpen	M	global	all	
Modem::IsoOpen	M	global	all	
Modem::Send	M	global	all	
<Client>::Receive	MB	global	Modem (all)	
Modem::Close	M	global	all	
Modem::GetCapability	M	global	all	
Modem::SetConfiguration	M	global	all	

6.10.3 Modem Data Structures

ModemType

Prototype

```
enum ModemType {PSTN};
```

Description

Defines the type of modem. (`PSTN` indicates a modem for use on Public Switched Telephone Networks.)

CommunicationSetup

Definition

```
struct PSTNSetup {
      long minDataRate;
      long maxDataRate;
      wstring<64> phoneNumber;
      boolean voice;
      boolean data;
      boolean fax;
      boolean SVD;
};

union CommunicationSetup switch (ModemType) {
   case PSTN: PSTNSetup     pstn;
};
```

Parameters

- `minDataRate` – minimum rate at which the client software element needs communication to be established (in bps). The value 0 indicates that communication must be performed at as high a rate as possible, without exceeding `maxDataRate`.
- `maxDataRate` – maximum data rate at which the client wants communication to be established (in bps). An application may need to specify this if it has to process incoming data, but is limited by its own computation, buffering... capabilities.
- `phoneNumber` – ASCII string of dialing characters:
  – when filled with a valid phone number, it indicates that the client application wants to initiate an outgoing call (*call mode*).
  – when left empty, it means that the client application is waiting for an incoming call *(answer mode*)
- `voice` – when `True`, sets the modem in voice mode.
- `data` – when `True`, sets the modem in data mode.
- `fax` – when `True`, sets the modem in fax mode.
- `SVD` – when `True`, sets the modem in simultaneous voice and data mode.

The `voice`, `data`, `fax`, `SVD` booleans can be interpreted by the Modem FCM in call mode (for instance, when an application wants to send a fax, the modem must be set in fax mode before phone number dialing). In call mode, *only one* of the `voice`, `data`, `fax`, `SVD` booleans can be set to `True` at the same time.

In answer mode, the Modem FCM is able to recognize the modulation scheme (during handshake, via ring indicator...), and thus switch into the recommended mode. However, the `voice`, `data`, `fax`, `SVD` booleans *must* have priority effect on the Modem FCM in answer mode (for instance to forbid any incoming faxes, or to only accept incoming voice calls...). In answer mode, several of the `voice`, `data`, `fax`, `SVD` booleans can be set to `True` at the same time in order to enable/disable reception of several incoming call types.

Description

Defines the communication parameters.

ModemCapabilities

Definition

```
struct PSTNCapabilities {
      long maxLineSpeed;
      long maxCommunicationNumber;
      boolean isoOpenSupported;
      boolean setConfigSupported;
};

union ModemCapabilities switch(ModemType){
   case PSTN: PSTNCapabilities   pstn;
};
```

Parameters

- `maxLineSpeed` – maximum line speed the FCM modem is able to support (in bps: e.g. 33600 bps for a V34 modem).
- `maxCommunicationNumber` – maximum number of connections between the client application and the Modem FCM that the Modem FCM can support simultaneously for each client (essentially depends on the hardware).
- `isoOpenSupported` – when `True` the modem supports the `Modem::IsoOpen` API.
- `setConfigSupported` – when `True` the modem supports the `Modem::SetConfiguration` API.

Description

Modem capabilities.

ModemDisconnection

Prototype

```
struct ModemDisconnection {
      long    cid;
};
```

Description

Used by the `Modem::disconnection` attribute.

ModemCallAccept

Prototype

```
struct ModemCallAccept {
      long    cid;
};
```

Description

Used by the `Modem::callAccept` attribute.

FileLoc

Prototype

```
enum FileLoc {START, MIDDLE, END};
```

Description

Indicates whether the message from a producer to a consumer is the first of a transfer (START), in the middle of a transfer (MIDDLE) or the last of a transfer (END). END is used if the transfer is accomplished in a single message.

6.10.4 Modem API

Modem::AsyncOpen

Prototype

```
Status Modem::AsyncOpen(
      in CommunicationSetup commInfo,
      out long cid,
      in short clientBufferSize,
      in OperationCode opCode,
      out short modemFcmBufferSize)
```

Parameters

- commInfo – setup information.
- cid – identifier of the connection. It allows starting several connections from a single software component and also permits matching a response with a request.
- clientBufferSize – indicates the maximum size (in bytes) of a message accepted by the requester. The Modem FCM will take this parameter into account during the sending of incoming transfers.
- opCode – operation code provided by the client that the Modem FCM will use to forward any incoming data. The client function identified by this operation code must be designed according to the <Client>::Receive API.
- modemFcmBufferSize – indicates the maximum size (in bytes) of a message accepted by the node where resides the Modem FCM. The client software element will take this parameter into account during the sending of outgoing transfers.

Description

This function allows a software element to open an asynchronous connection with the required configuration parameters. Each Modem::AsyncOpen operation allows the Modem FCM to know which function to call in order to forward data to its client.

Error codes

- Modem::ENETWORK – the modem is no longer connected to the external network .
- Modem::EBUSY – remote modem is busy.
- Modem::ESETUP – communication with the remote modem cannot be established with the requested communication setup because the Modem FCM does not support this configuration, or because line conditions do not allow it.
- Modem::EINVALID_MODE – connection is impossible because Modem FCM does not support the required mode (Fax, Voice, data or SVD mode).
- Modem::ENUM_CONN – maximum number of opened connections is reached for this FCM.
- Modem::EFORBIDDEN – this phone number is not authorized to be called.

Modem::IsoOpen

Prototype

```
Status Modem::IsoOpen(
```

```
      in CommunicationSetup commInfo,
      out long cid,
      in FcmPlug modemFcmOutputPlug,
      in FcmPlug modemFcmInputPlug)
```

Parameters

- `commInfo` – setup information.
- `cid` – identifier of the connection. It allows starting several connections from a single software component and also permits matching a response with a request.
- `modemFcmOutputPlug` – an output plug of the Modem FCM.
- `modemFcmInputPlug` – an input plug of the Modem FCM.

Description

This function allows a software element to open an isochronous connection between the client software element plug and the Modem FCM plug, relying on Stream Manager facilities. Before calling this `Modem::IsoOpen` function, the client software element should first:

- use the `Fcm::GetPlugCount` and `Dcm::GetPlugStatus` methods to know which plug of the Modem FCM could be used for the connection between itself (client) and the modem.

- use the `StreamManager::FlowTo` method to create an isochronous stream connection between itself (client) and the modem.

Once the plugs have been connected by the Stream Manager, the client can call the `Modem::IsoOpen` function to get an identifier for the modem connection.

This API is optional. Its support can be verified by the `Modem::GetCapability` API.

Error codes

- `ENOT_IMPLEMENTED` – the modem does not support isochronous connection.
- `Modem::ENETWORK` – the modem is no longer connected to the external network .
- `Modem::EINVALID_PLUG` – at least one of the specified plugs is invalid
- `Modem::EBUSY` – remote modem is busy.
- `Modem::ESETUP` – the Modem FCM does not support the communication setup.
- `Modem::ENUM_CONN` – maximum number of opened connections is reached for this FCM.
- `Modem::EFORBIDDEN` – this phone number is not authorized to be called.

Modem::Send

Prototype

```
Status Modem::Send (
      in long cid,
      in FileLoc where,
      in sequence<octet> data)
```

Parameters

- `cid` – identifier of the connection between the client application and the Modem FCM (issued by a client application from a previous `AsyncOpen` or `IsoOpen` call).
- `where` – informs the Modem FCM that this message contains the first, the last or a middle segment of the data to be transferred.
- `data` – contains a part of (a multi-segment transfer) or the entire data to be transferred.

Description

This function allows a client software element to send data to the Modem FCM.

Error codes

- `Modem::ENETWORK` – the modem is no longer connected to the external network .
- `Modem::ESIZE` – the data exceeds the size of the buffer in the receiver. The receiver has not received or processed the data. It is left to the implementation how the sender reacts to this status.
- `Modem::EFAILED` – the receiver has aborted the transfer of the current sequence of data transfers. The sender shall abort the transfer of the current sequence.
- `Modem::ECID` – `cid` is not correct or unknown.

<Client>::Receive

Prototype

```
Status <Client>::Receive (
      in long cid,
      in FileLoc where,
      in sequence<octet> data)
```

Parameters

- `cid` – identifier of the connection between the client application and the Modem FCM (issued by a client application from a previous `AsyncOpen` or `IsoOpen` call).
- `where` – informs the software element client that the message contains the first, the last or a middle segment of the data to be transferred.
- `data` – contains a part (a multi-segment transfer) or the entire data transferred for the connection identified by the `cid` parameter.

Description

`<Client>::Receive` is implemented in the client software element. This function allows the Modem FCM to forward any incoming data to the client software element.

Error codes

- `Modem::ESIZE` – the data exceeds the size of the buffer in the receiver. The receiver has not received or processed the data. It is left to the implementation how the sender reacts to this status.
- `Modem::EFAILED` – the receiver has aborted the transfer of the current sequence of data transfers. The sender shall abort the transfer of the current sequence.
- `Modem::ECID` – incorrect or unknown `cid`

Modem::Close

Prototype

```
Status Modem::Close(in long cid)
```

Parameters

- `cid` – identifier of the connection between the client software element and the Modem FCM that has to be removed.

Description

This function allows a software element to close a modem connection. If an isochronous connection has been opened, the client application should drop the isochronous stream between itself and the Modem FCM using the appropriate Stream Manager method.

Error codes

- `Modem::ECID` – `cid` is not correct or unknown.

Modem::GetCapability

Prototype

```
Status Modem::GetCapability(
      out sequence<ModemType> typeList,
      out sequence<ModemCapabilities> capabilityList)
```

Parameters

- `typeList` – list of types supported by the queried Modem FCM.
- `capabilityList` – modem features corresponding to each different modem type of `typeList`.

Description

This function allows a software element to get all modem types supported by the Modem FCM, and the capabilities associated with each type.

Modem::SetConfiguration

Prototype

```
Status Modem::SetConfiguration   (
      in long cid,
      in CommunicationSetup commInfo)
```

Parameters

- `cid` – connection identifier identifying the connection for which the configuration has to be changed.
- `commInfo` – modem setup information

Description

This function allows a software element to set a modem and/or a connection into a particular configuration (e.g., dynamically change the data transfer rate during a communication, switch from data mode to voice mode during a multi-player game communication...)

However, to switch from *call mode* to *answer mode* (or conversely), the client application must close the connection and re-open a new one.

This API is optional. Its support can be verified by the `Modem::GetCapability` API.

Error codes

- `ENOT_IMPLEMENTED` – the modem does not support this function.
- `Modem::ECONN` – the connection cannot be maintained with this new configuration. The setup change request will not be taken into account, and old parameters will be kept. It is left up to the application to resume the connection.
- `Modem::ECID` – `cid` is not correct or unknown.

6.10.5 Modem Notification Attributes

Modem::disconnection

Attribute
`ModemDisconnection disconnection`

Description

Occurrence of a network disconnection on the connection specified by the `cid` value. Only useful with comparator `ANY`.

Modem::callAccept

Attribute
`ModemCallAccept callAccept`

Description

Occurrence of an incoming call on the connection specified by the `cid` value. Only useful with comparator `ANY`.

6.11 Web Proxy FCM

The Web Proxy FCM offers sharable access to the Internet. It supports Internet protocols (for example HTTP messages) between a Web client and a Web server.

The software element client has only to find a Web Proxy FCM in the HAVi network (using the Registry service) and call the corresponding functions to initiate its Web connection. It can then send or receive data according to an “Internet application protocol” (HTTP for example).

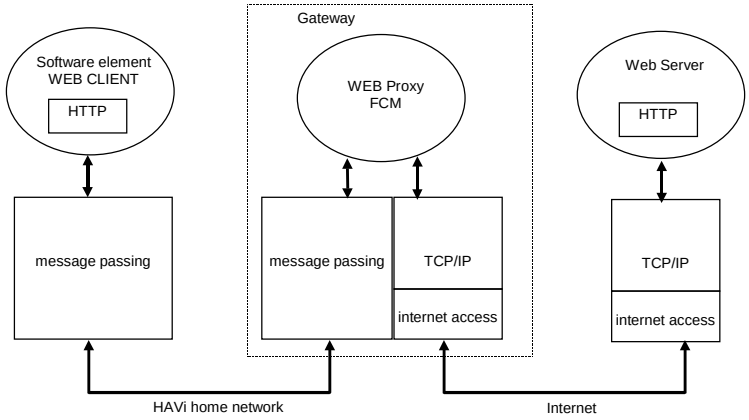

Figure 37. Web Proxy Communication

The Web Proxy FCM has to support (at least one of) the commonly used Internet application protocols like HTTP, FTP, NNTP, SNMP, POP and IMAP. Its API is sufficiently flexible to allow the extension with future client/server protocols that run over an IP stack.

6.11.1 Overview

6.11.1.1 The Web Gateway

The Web gateway is any BAV, IAV or FAV device within the HAVi network. Figure 37 shows the case where the Web Proxy FCM is on the gateway itself. However, the HAVi architecture allows the Web Proxy FCM to be installed outside the gateway device (i.e., the gateway could be a BAV device).

The Web Proxy FCM can be viewed as a general purpose proxy or user agent. Consequently the current specification could be extended by defining constraints that provide facilities according to the specification of particular application protocols. For example, a possible HTTP constraint would be that the Web Proxy FCM interpret the URL (within HTTP requests) and use the protocol corresponding to the scheme indicated in the URL.

The gateway has to contain the IP stack, according to IETF RFC 1122 and RFC 1123, and means to access the Internet. Access (see Figure 37) could be:

- a link to a service provider through the PSTN or the ISDN (using a telephone modem)
- a link to service provider through a cable network (using a cable modem)

The Web Proxy FCM is associated with the gateway and, of course, a DCM. This DCM has to

offer a GUI through DDI or an uploadable havlet. This GUI allows a user to configure the gateway (phone number of web service provider, modem speed for example).

6.11.1.2 The Web Client

The Web client is the software element that reaches a Web server using one of the supported application protocols. It could be, for example, a Web browser or the DCM Manager (when it has to upload a DCM).

6.11.1.3 Web Proxy FCM Protocol

6.11.1.3.1 Multiple Web Transactions

The Web Proxy FCM offers an API that allows one or more client software elements to send transactions to a Web server. Moreover a client could manage several consecutive transactions.

To use the service of the Web Proxy FCM, the Web client opens a connection with the FCM according to the desired application protocol (HTTP for example). It can then communicate: send and receive (HTTP transactions for example). Finally it closes the connection.

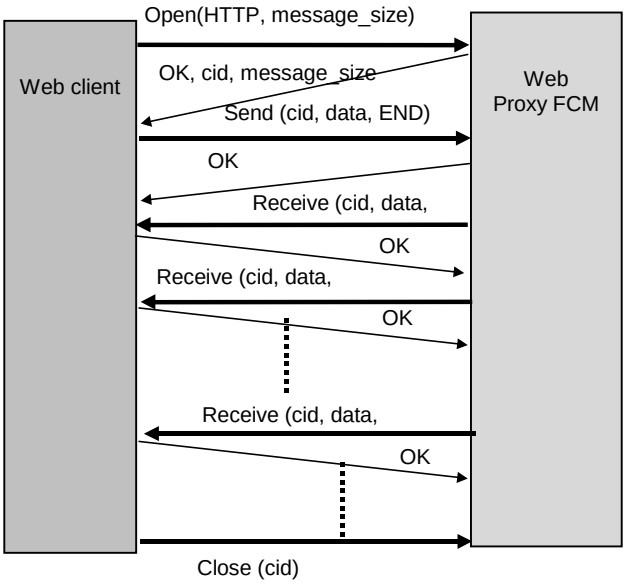

Figure 38. Web Proxy and a Web Client Communication

In Figure 38, the Web client opens a connection with the Web Proxy FCM to use it as a proxy HTTP agent. To open a connection the Web client has to provide the Internet application protocol type, the address of the Web server and the maximum message buffer size (taking into account the message passing protocol overhead – see section 5.14).

The Web Proxy FCM establishes the Internet connection (contacts its service provider if needed). With the response of the `Open` request the Web Proxy FCM provides an identifier for the connection (cid) and also its maximum message buffer size (taking into account the message passing protocol overhead – see section 5.14).

Once the connection is established the Web client can start to use the link according to its protocol. In Figure 38 the Web client sends a HTTP request (which fits within one HAVi message). The Web Proxy FCM receives this message, performs a TCP connection[4] with the HTTP server and then forwards the request to the server, waits for the response and returns it using one or more messages according to the response size.

The connection is closed by the Web client.

In the case of a lost message, the Web Proxy FCM will be blocked until reception of the response and the client will also be blocked until the reception of the next message. In this case it is the responsibility of the client to close the connection after a time out fixed by the client.

The way to map a client with the HTTP transaction is vendor dependent. (For example, the SEID and connection identifier of the Web client could be mapped with a TCP connection[5].)

6.11.1.4 Application Protocols

The Internet application protocol may be one of the following:

- HTTP1.1
- FTP
- SMTP
- POP3
- IMAP4
- NNTP

The API offered by the Web Proxy FCM is sufficiently flexible to accept new protocols. A "get capability" service allows the client to determine the protocols supported by the Web Proxy FCM.

6.11.1.5 Application Protocol Constraints

This section gives, for each application protocol supported by the Web Proxy FCM, a list of constraints, rules and actions the Web Proxy FCM must support.

6.11.1.5.1 HTTP Constraints

The Web Proxy FCM acts as an HTTP proxy server and interprets the URL and reacts in conformance with the RFC 1738.

6.11.1.5.2 FTP Constraints

No constraints are defined.

6.11.1.5.3 SMTP Constraints

No constraints are defined.

[4] It is not necessary to open a TCP connection for each HTTP request.

[5] Typically the TCP connection identifier could be the socket ID using the socket API.

6.11.1.5.4 IMAP Constraints

No constraints are defined.

6.11.1.5.5 POP Constraints

No constraints are defined.

6.11.1.5.6 NNTP Constraints

No constraints are defined.

6.11.2 Web Proxy Services

Service	Comm Type	Locality	Access	Resv Prot
WebProxy::Open	M	global	all	
WebProxy::Close	M	global	all	
WebProxy::Send	M	global	all	
<Client>::Receive	MB	global	WebProxy (all)	
WebProxy::GetCapability	M	global	all	

6.11.3 Web Proxy Data Structures

FileLoc

Prototype

```
enum FileLoc { START, MIDDLE, END};
```

Description

Indicates whether the message from a producer to a consumer is the first of a transfer (`START`), in the middle of a transfer (`MIDDLE`) or the last of a transfer (`END`). `END` is used if the transfer is accomplished in a single message.

InternetProtocolType

Prototype

```
enum InternetProtocolType { HTTP, FTP, SMTP, POP3, IMAP4,
NNTP};
```

Description

The list of Internet application protocols the Web Proxy FCM may support.

WebAddressType

Prototype

```
enum WebAddressType { DN, IP, LOCAL};
```

Description

Indicates the type of an Internet host address: either a domain name or an IP address, or the device associated with the Web Proxy FCM (i.e., the gateway in Figure 37).

WebAddressTypeIP

Prototype

```
enum WebAddressTypeIP { IPV4, IPV6};
```

Description

Indicates the type of an IP address: either an IPv4 address (as defined in section 3.1 of IETF RFC 1738) or an IPv6 address (as defined in IETF RFC 2373).

WebAddressIP

Prototype

```
union WebAddressIP switch (WebAddressTypeIP)
{
   case IPV4:    octet[4]   v4Address;
   case IPV6:    octet[16]  v6Address;
};
```

Description

Indicates an IP address in either IPv4 address format or IPv6 address format.

WebAddressName

Prototype

```
typedef sequence <octet, 255> WebAddressName;
```

Description

Indicates a domain name (as defined in section 3.1 of IETF RFC 1738).

WebAddress

Prototype

```
union WebAddress switch(WebAddressType)
{
   case  DN:     WebAddressName   name;
   case  IP:     WebAddressIP     address;
   case  LOCAL:  ;
}
```

Description

Indicates an Internet host address which will be formatted either as an IP address or a domain name. `LOCAL` refers to the device associated with the Web Proxy FCM (i.e., the gateway in Figure

37).

WebProxyDisconnection

Prototype

```
struct WebProxyDisconnection {
      long    cid;
};
```

Description

Used by the `WebProxy::disconnection` attribute.

6.11.4 Web Proxy API

WebProxy::Open

Prototype

```
Status WebProxy::Open(
      in InternetProtocolType protocol,
      in short clientBufferSize,
      in OperationCode opCode,
      in WebAddress address,
      in uint portNumber,
      out long cid,
      out short proxyBufferSize)
```

Parameters

- `protocol` – the application protocol for the session the requester wants to open. The Web Proxy FCM shall apply some well-known constraints according to this protocol described in section 6.11.1.5.
- `clientBufferSize` – indicates the maximum size (in bytes) of a message accepted by the requester. The Web Proxy FCM will take this parameter into account during the sending of the responses to the client.
- `opCode` – this is the operation code the Web Proxy FCM will use to forward to the client an incoming response. The client function identified by this operation code must be designed according to the `<Client>::Receive` API.
- `address` – the address of the host with which the caller wants to communicate. If the address type is `LOCAL` and the Web Proxy FCM does not support this option then the `WebProxy::EADDRESS` error code shall be returned.
- `portNumber` – the port number for the underlying UDP or TCP communication. If this value is 0 then the port number will be the default port number associated with `protocol`.
- `cid` – the identifier of this connection with the Web Proxy FCM. It allows several connections from the same software component client and also permits matching a response with a request.
- `proxyBufferSize` – indicates the maximum size (in bytes) of messages accepted by the Web Proxy FCM. The Web client will take this parameter into account during the sending of the requests.

Description

This function allows a software element client (or Web client) to open a connection with a Web Proxy FCM. This connection is characterized by the application protocol to be used over the

connection.

Error codes

- `ERESOURCE_LIMIT` – resource allocation error
- `WebProxy::ENUM_CONN` – maximum number of opened connections is reached for this FCM
- `WebProxy::ENETWORK` – the Web Proxy FCM is no longer connected to the external network
- `WebProxy::EPROTOCOL` – the protocol type is not supported by the Web client
- `WebProxy::EADDRESS` – the address is unknown

WebProxy::Close

Prototype

```
Status WebProxy::Close(in long cid)
```

Parameters

- `cid` – the identifier of this connection with the Web Proxy FCM.

Description

This function is used to close a connection with a Web Proxy FCM.

Error codes

- `WebProxy::ECID` – The `cid` is unknown.

WebProxy::Send

Prototype

```
Status WebProxy::Send(
      in long cid,
      in FileLoc where,
      in sequence<octet> webData)
```

Parameters

- `cid` – the identifier of the connection between the Web client and the Web Proxy FCM. Issued to the Web client by a previous `WebProxy::Open` call.
- `where` – informs the Web Proxy FCM that this message contains the first, the last or a middle segment of the data to be transferred.
- `webData` – contains a part (a multi-segment transfer) or the entire request according to the application protocol used on the connection identified by the `cid` parameter.

Description

This function allows a software element client (or Web client) to send a request to a Web server according to the application protocol (HTTP for example).

Error codes

- `WebProxy::ESIZE` – the data exceeds the size of the buffer in the receiver. The receiver has not received or processed the data. It is left to the implementation how the sender reacts to this status.
- `WebProxy::ENETWORK` – the Web Proxy FCM is no longer connected to the external network..
- `WebProxy::EFAILED` – the receiver has aborted the transfer of the current sequence of data

transfers. The sender shall abort the transfer of the current sequence.
- `WebProxy::ECID` – the `cid` is unknown.

<Client>::Receive

Prototype

```
Status <Client>::Receive(
      in long cid,
      in FileLoc where,
      in sequence<octet> webData)
```

Parameters

- `cid` – the identifier of the connection between the Web client and the Web Proxy FCM. Issued to the Web client by a previous `Open` call.
- `where` – informs the Web client that the message contains the first, the last or a middle segment of the data to be transferred.
- `webData` – contains a part (a multi-segment transfer) or the entire response according to the application protocol used on the connection identified by the `cid` parameter.

Description

`<Client>::Receive` is implemented in the client and allows the Web Proxy FCM to forward to the client an incoming response according to the application protocol (HTTP for example).

Error codes

- `WebProxy::ESIZE` – the data exceeds the size of the buffer in the receiver. The receiver has not received or processed the data. It is left to the implementation how the sender reacts to this status.
- `WebProxy::EFAILED` – the receiver has aborted the transfer of the current sequence of data transfers. The sender shall abort the transfer of the current sequence.
- `WebProxy::ECID` – the `cid` is unknown.

WebProxy::GetCapability

Prototype

```
Status WebProxy::GetCapability(
      out sequence<boolean> protocolList,
      out sequence<boolean> typeList)
```

Parameters

- `protocolList` – the list of Web application protocols which are available through this FCM. The i^{th} member of `protocolList` indicates the availability of the protocol identified by `InternetProtocolType` value *i*. The safe parameter size is 32 `boolean` values.
- `typeList` – the list of Web address types that are supported by this FCM. The i^{th} member of `typeList` indicates the availability of the address type identified by `WebAddressType` value *i*. The safe parameter size is 32 `boolean` values.

Description

This function permits a Web client to discover the protocols supported by the Web Proxy FCM.

6.11.5 Web Proxy Notification Attributes

WebProxy::disconnection

Attribute
`WebProxyDisconnection disconnection`

Description

Occurrence of a disconnection on the connection specified by the `cid` value. Only useful with comparator `ANY`.

7 HAVi Java API Description

7.1 Overview

To allow flexible and future-proof CE platforms, HAVi supports the uploading of Device Control Modules, Application Modules and havlets on HAVi FAV devices. The uploadable entities are written in Java bytecode and an FAV supports a Java virtual machine on which these entities can run. This chapter describes the definitions necessary to guarantee interoperability with respect to the uploading and execution of Java bytecode.

7.2 Profiles

HAVi FAV nodes support the uploading of different types of HAVi Java entities. Each FAV must be able to host DCMs of BAV devices, and so must be able to upload and execute DCM code units. Moreover, each FAV itself decides whether it uploads Application Modules or havlets. To guarantee that a HAVi Java entity shall be able to execute on each HAVi FAV to which it may be uploaded, two FAV profiles are defined indicating which classes and packages an FAV must support.

- Profile #1: Packages and classes that are (may be) used in DCM code units and Application Modules code units, and therefore must be supported by every FAV.

- Profile #2: Packages and classes that are (may be) used by havlets and therefore, need to be supported by each FAV that uploads and execute havlets. This set of packages and classes is an extension of the set of Profile #1.

Note that whether an FAV uploads havlets is a decision it makes on its own and is not influenced or ordered by other HAVi nodes in the network. Therefore, it is not necessary for other nodes, or for (DCM/Application Module/havlet) code units to know whether an FAV is of profile #1 or #2.

The uploadable code unit shall not include classes which are in the org.havi package name-space or any package name-space which begins with 'org.havi'. FAVs shall not execute any bytecode contained in such classes. The mechanism by which they achieve this (and hence the time when rejection of the class occurs) is implementation dependent. It is recommended that platforms consider using the checkPackageDefinition method of java.lang.SecurityManager to accomplish this.

7.2.1 Java API Referencing Rules

There are a number of situations in this specification where there are references from Java APIs which are mandatory in the HAVi specification to Java APIs which are not mandatory in the HAVi specification. These include the following :

- Methods where at least one of the arguments is a class or interface which is not mandatory in the HAVi specification.

- Methods whose return value is specified as a class or interface which is not mandatory in the HAVi specification.

- Methods which throw an exception which is not mandatory in the HAVi specification.

- Fields whose type is a class or interface which is not mandatory in the HAVi specification.

In these situations, the HAVi specification does not require the method or field concerned to be present in implementations.

Their presence or otherwise is a technical and licensing issue for implementations. Implementations may include more than is specified here. However DCM code units, Application Module code units, and havlet code units are not compliant if they require more classes or interfaces than defined in this specification.

7.2.2 Profile #1: DCMs and Application Modules

The following packages and classes must be supported by each FAV and may be used in DCM and Application Module code units:

- java.lang, as defined in the Java 1.1 Core API (reference [8])
- java.util, as defined in the Java 1.1 Core API (reference [8])
- org.havi.constants, as defined in Appendix A
- org.havi.types, as defined in Appendix A
- org.havi.system, as defined in Appendix A
- org.havi.Iec61883, as defined in Appendix A
- org.havi.fcm.*, as defined in Appendix A
- java.io as defined in the Java 1.1 Core API (reference [8]) apart from the following classes File, FileInputStream, FileNotFoundException, FileOutputStream, FileReader, FileWriter, FilenameFilter and RandomAccessFile. These named classes are not mandatory in this specification.
- java.net.URL, MalformedURLException as defined in the Java 1.1 Core API (reference [8])

The external forms used by objects implementing java.io.Serializable is not fixed by this specification or any of its referenced specifications. Hence there is no requirement for the classes java.io.ObjectInputStream and java.io.ObjectOutputStream to be inter-operable between HAVi implementations.

Support of `java.net` package is required for at least one implementation dependent protocol for use with instances of the java.net.URL class. The methods Class.getResource() and ClassLoader.getResource() shall return instances of java.net.URL using this protocol when used to access files carried in the JAR file of an uploaded code unit.

The method URL.getContent shall work as specified in its specification even though the reference to the URLConnection is not required to be valid. Except as specified in the list below, this method shall return instances of java.io.InputStream for all content types.

- For images, java.awt.image.ImageProducer shall be returned

Moreover, to upload and install code units (see next section) an FAV may need to support (parts of) the following packages:

- java.util.zip, as defined in the Java 1.1 Core API (reference [8])

An FAV only needs to provide parts of these packages to properly implement the uploading and installation of HAVi code units, it does not need to provide these classes to uploaded Java entities. So, writers of DCM and Application Module code units shall not rely on the availability of the java.io and java.util.zip packages.

All interfaces added as proprietary extensions to org.havi.* shall have default (package) access. All classes, methods and fields added as proprietary extensions to org.havi.* shall have default (package) or private access.

As long as the defined contracts (specification) are respected and kept, it is an allowable option to override protected, public and package level methods in the org.havi.* package using the standard java inheritance conventions.

The following restrictions apply to the use of the java.lang package by uploaded code units. Uploaded code units shall not violate these restrictions. The behavior of FAVs should an uploaded code unit violate these restrictions is not specified. These restrictions do not remove any class or method signatures concerned from the platform.

- The following methods shall not be called:
  - Runtime.exec()
  - Runtime.load()
  - Runtime.loadLibrary()
  - Runtime.runFinalizersOnExit()
  - System.exit()
  - System.load()
  - System.loadLibrary()
  - System.runFinalizersOnExit()
  - Thread.stop()
  - Thread.suspend()
  - Thread.resume()
- The following fields shall not be used:
  - System.in
- Methods in the following classes shall not be called:
  - java.lang.Process
- Uploaded code units shall be able to use:
  - System.out
  - System.err
  - Runtime.traceInstructions()
  - Runtime.traceMethodCalls()

for debugging without any adverse effects to the code unit. The output shall not be visible to normal end users and shall not conflict with any other API or feature of the platform. It is an allowed implementation option to not generate any output. It is not an allowed implementation to stall or block until some implementation specific debugging device is connected.

- The java.lang.Compiler class and following methods shall be taken as hints from an application to the system however there is no guarantee of what happens:

  - Runtime.gc()
  - System.gc()

- The System.setProperties() and System.setSecurityManager() methods will always throw an exception when called by uploaded code units.

- SecurityManager.checkCreateClassLoader() shall always throw a SecurityException if an application attempts to create its own subclass of java.lang.ClassLoader.

7.2.3 Profile #2: Havlets

Besides the packages defined in Profile #1, the following packages must be supported by each FAV that uploads havlets and may be used in havlet code units:

- A subset of Java AWT as defined in section 8.2.2

- org.havi.ui and org.havi.ui.event, as defined in Appendix A

7.3 Mapping HAVi IDL to Java

7.3.1 Introduction

The HAVi specification uses a subset of IDL to represent data types, structures and API's (IDL operations). The intent is to provide a normative way of representing the API's. The HAVi Messaging System will send data in big-endian order and the mapping of data types is based on CDR (Common Data Representation) from GIOP (General Inter Orb Protocol) version 1.1 as described in section 3.2.3.4.

This section explains the rules applied to derive classes that form the basis of the HAVi Java APIs. The rules are mostly derived from Object Management Group's IDL to Java mapping document. However, to keep with the requirement that the HAVi Java APIs be of a small memory foot-print and provide high performance, the rules have been slightly modified. This requirement is consistent with the requirements of consumer electronic devices.

7.3.2 const

An IDL const with constructed type is mapped to a public interface with the same name as the type of the constant with the prefix "Const". The value of the const is mapped to a public static final field in the interface with the same name as the constant. All tables with constant values in the annex of this specification are mapped to appropriate "Const" interfaces.

An example of mapping a `const` of a constructed type to Java is given below:

IDL	JAVA

`const TypeX A = 0x0005;`
`const TypeX B = 0x0008;`
	public interface ConstTypeX {
	public static final TypeX A = 0x0005;
	public static final TypeX B = 0x0008;
	}

7.3.3 Basic Types

7.3.3.1 boolean

The IDL `boolean` type correspond to the boolean Java type.

7.3.3.2 char, wchar and octet

The IDL type `char` is mapped to the Java type byte.

The IDL type `wchar` is mapped to the Java type char. Since HAVi states that `wchar` size is 2 bytes long and is in the UNICODE char set then the mapping is natural. HAVi recommends using the IDL `wchar` type for printable characters.

The IDL type `octet` is mapped to the Java type byte**.**

7.3.3.3 string and wstring

OMG IDL defines the string type `string` to be consisting of all possible 8-bit quantities except null. A `string` is similar to a sequence of `char`. Hence the mapping to Java is a sequence of bytes.

The IDL type `wstring` is mapped to the Java.lang.String. A `wstring` in HAVi only contains UNICODE characters (two bytes per character)

7.3.3.4 Integers

The IDL integer types are mapped to the Java integer types as the followings:

IDL	Java
`short`	short
`unsigned short`	short
`long`	int
`unsigned long`	int
`long long`	long
`unsigned long long`	long

As the `unsigned` qualifier does not exist in Java, the programmer will have to take care of comparisons, which will involve an IDL integer.

The corresponding Java class (Integer, Long) will be used as help for unsigned comparisons.

7.3.3.5 Floating Points

The IDL floating-point types (`float` and `double`) are mapped with the corresponding Java floating point types.

7.3.4 Constructed Types

The IDL `struct` and `union` types defined in HAVi are mapped to class definitions in Java. These class definitions can be found in org.havi.types. These classes extend either the HAVi defined HaviObject abstract class or HaviImmutableObject abstract class.

The `HaviObject` class extends the java.lang.Object class and implements both of the org.havi.Marshallable interface and the java.lang.Cloneable interface. The HaviObject class implements an equals method, a hashCode method, a marshal method, an unmarshal method and a clone method as abstract methods.

The HaviImmutableObject class extends the HaviObject class and implements an unmarshal method as a final method which always throw the HaviUnmarshallingException exception.

If the classes extend the HaviObject class, then they implement an equal method, a hashCode method, a marshal method, an unmarshal method and a clone method as concrete methods.

If the classes extend the HaviImmutableObject class, then they implement an equals method, a hashCode method, a marshal method, and a clone method as concrete methods.

Also the classes must provide a constructor for constructing an instance of the object from a HaviByteArrayInputStream. This is further explained in the Marshalling and Unmarshalling section. Classes extending HaviImmutableObject do not permit any methods that change the member values once the object is constructed. That is, there must be no accessor methods to change any value of the object.

The equals method concerns semantic equality of objects, i.e., equality of all fields. If the argument of the equals method is null, the equals method shall return false. The hashCode method shall return a hash code value for the object and shall provide the same contract as specified in the description of hashCode method in java.lang.Object. The clone method of the class which extends the HaviObject class shall provide recursive cloning of objects and return the clone object, i.e., if object X refers to an object Y, a clone of X refers to a clone of Y. The clone method of the class which extends the HaviImmutableObject class shall return the object itself for which the clone method is called. Marshalling and unmarshalling are explained in Section 7.3.7.

7.3.4.1 enum

The IDL `enum` type is mapped to a public Java interface. The Java interface name is the name of the IDL `enum` type with the prefix “Const”. Each `enum` value corresponds to a public static final field. If no specific value is provided for the enum then the value assigned will start from 0 and will be incremented in units of 1 for succeeding values.

An example of mapping an `enum` to Java is given below:

IDL	JAVA
`enum TypeX { A, B, C }`	public interface ConstTypeX{ public static final int A =0, B=1,

	C=2; }

7.3.4.2 struct

The IDL `struct` type is mapped to a final Java class. The Java class name is the name of the IDL `struct` type. Each `struct` member corresponds to a pair of an accessor method and a modifier method for an internal private field which keeps the member value. The Java class provides a constructor to initialize the struct object and additionally a null constructor (the struct fields could be updated after the object creation) if it extends from HaviObject.

In addition all classes defined for `struct` must extend from either the abstract HaviObject class or the abstract HaviImmutableObject class. They must implement the methods equals, hashCode, marshal and clone. In addition, the unmarshal method must be implemented if the classes defined for struct extends HaviObject. Also the classes must provide a constructor for constructing an instance of the object from a HaviByteArrayInputStream. This is further explained in section 7.3.7 on marshalling and unmarshalling.

An example of mapping a `struct` to Java is given below:

IDL	JAVA

`struct TypeStruct`
`{`
`  AType A;`
`  Btype B;`
`  Ctype C;`
`};`
	public final class TypeStruct extends HaviObject {
	  private AType A;
	  private Btype B;
	  private Ctype C;
	
	  public final TypeStruct(){}
	  public final TypeStruct(AType _A, Btype _B, Ctype _C){
	  A = _A;
	  B = _B;
	  C = _C;
	  }
	
	  public final TypeStruct(HaviByteArrayInputStream hi)
	  throws HaviUnmarshallingException {
	  …
	  }
	
	  public void marshal(HaviByteArrayOutputStream ho)
	  throws HaviMarshallingException {
	  ...
	  }
	
	  public void unmarshal(HaviByteArrayInputStream hi)
	  throws HaviUnmarshallingException {
	  …
	  }
	
	  public boolean equals(Object o) {
	  ....
	  }
	
	  public int hashCode() {
	  …
	  }
	
	  public Object clone() {
	  ...
	  }
	
	}

The behavior of accessor methods getting each struct member value, modifier methods setting each struct member value, and constructors initializing instances shall be as follows:

- If the Java class extends the HaviObject class
  - If the field is a class other than subclasses of the HaviImmutableObject class, a reference to the object is got/set.
  - If the field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.
  - If the field is of the basic data type, the basic data is got/set by value
- If the Java class extends the HaviImutableObject class
  - If the field is a class other than subclasses of the HaviImmutableObject class, a deep copy of the object is created and got/set.
  - If the field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.
  - If the field is of the basic data type, the basic data is got/set by value

7.3.4.3 union

The IDL `union` is mapped to a final Java class with the same name that has:

- a default constructor (only if extending from HaviObject)
- one constructor method for each branch
- an accessor method for the discriminator, named `getDiscriminator()`
- an accessor method for each branch
- a modifier method for each branch (only if extending from HaviObject)
- a modifier method for each branch that has more than one case label (only if extending from HaviObject)
- a default modifier method if needed (only if extending from HaviObject)

The normal name conflict resolution rule is used (prepend an "_") for the discriminator if there is a name clash with the mapped union type name or any of the field names.

The branch accessor and modifier methods are overloaded and named after the branch. Accessor methods shall raise the HaviUnionException if the expected branch is not set.

One modifier method exists for each member accepting as unique parameter the member's value; thus, the discriminator is automatically set to a legal value for that union member. One accessor method exists for each union member. It will return the value of the current member. Attempt to access a member which is not the current one (regarding the discriminator value) will generate an exception. The member's constructors initialize the discriminator to a specified value corresponding to a legal value (in case of unique non-default case label, then that label is used as the implicit discriminator). The default class constructor does not initialize the union. Thus it is necessary to use a member modifier method or member constructor method before using any accessor method.

All the classes mapped from unions will have a set of constructors which can initialize the class to any branch of the union. These constructors will have a parameter called switchType to specify the branch.

In addition all classes defined for `union` must extend either the abstract HaviObject class or the abstract HaviImmutableObject class. They must implement the abstract methods equals, hashCode, marshal and clone. In addition, the `unmarshal` method must be implemented if the classes defined for union extends HaviObject.. All classes must provide a constructor for constructing an instance of the object from a HaviByteArrayInputStream. This is further explained in section 7.3.7 on marshalling

and unmarshalling.

An example of mapping a `union` to Java is given below:

IDL	JAVA

`union TypeUnion switch (long) { case 1: AType A; case 2: Btype B; case 3: default:Ctype C; };`	public final class TypeUnion extends HaviObject {
	
	private int discriminator; // the discriminator
	
	private AType A;
	private Btype B;
	private Ctype C;
	
	public int getDiscriminator ( ) throws HaviUnionException{...}
	
	public TypeUnion (HaviByteArrayInputStream hi)
	throws HaviUnmarshallingException { ... }
	
	public TypeUnion (int switchType, AType _A)
	throws HaviInvalidValueException { ... }
	public void setA (AType _A)
	throws HaviInvalidValueException { ... }
	public Atype getA ( )
	throws HaviUnionException { ... }
	
	public TypeUnion (int switchType, BType _B)
	throws HaviInvalidValueException { ... }
	public void setB (Btype _B)
	throws HaviInvalidValueException { ... }
	public BType getB ( )
	throws HaviUnionException { ... }
	
	public TypeUnion (int switchType)
	throws HaviInvalidValueException { ... }
	
	public TypeUnion (int switchType, CType _C)
	throws HaviInvalidValueException { ... }
	public void setC(Ctype _C)
	throws HaviInvalidValueException { ... }
	public CType getC ( )
	throws HaviUnionException { ... }
	
	public boolean equals (Object o) {
	....
	}
	
	public int hashCode() {
	…
	}
	
	public void marshal (HaviByteArrayOutputStream hbaos)
	throws HaviMarshallingException {
	....
	}
	
	public void unmarshal(HaviByteArrayInputStream hbais)
	throws HaviUnmarshallingException {
	…
	}
	
	public Object clone ( ) {
	....
	}

	}

The behavior of accessor methods getting each union member value, modifier methods setting each union member value, and constructors initializing instances shall be as follows:

- If the Java class extends the HaviObject class
  - If the field is a class other than subclasses of the HaviImmutableObject class, a reference to the object is got/set.
  - If the field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.
  - If the field is of the basic data type, the basic data is got/set by value
- If the Java class extends the HaviImutableObject class
  - If the field is a class other than subclasses of the HaviImmutableObject class, a deep copy of the object is created and got/set.
  - If the field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.
  - If the field is of the basic data type, the basic data is got/set by value

7.3.4.4 sequence

The IDL `sequence` type is mapped to a Java array of the mapped type. Bounded sequences imply a bound check during marshalling of IDL operation parameters. Holder classes (explained later) for the sequences are defined by using the same name as the type of the sequence with "SeqHolder" appended to it. However, `sequence`<octet> shall be mapped into HaviByteArrayOutputStream if it needs to be marshalled using CDR. And `sequence`<octet> shall be mapped into HaviByteArrayInputStream if it needs to be unmarshalled using CDR (See the section 7.3.7). Otherwise `sequence` <octet> shall be mapped into byte[].

7.3.4.5 array

The IDL `array` type is mapped to the Java array. Bounded arrays imply a bound check during marshalling and unmarshalling of IDL operation parameters.

7.3.4.6 typedef

Java does not have a typedef construct.

7.3.4.6.1 Simple IDL Types

Any `typedef` that is a type declaration for a simple type is mapped to the original (mapped type) everywhere the `typedef` type appears. Thus an IDL construct `typedef ushort ElementId` does not generate a corresponding ElementId class. Wherever `ElementId` appears in the IDL it is assumed to be replaced with `ushort` and the appropriate mapping rules are further applied.

7.3.4.6.2 Complex IDL Types

Typedefs for non-arrays and sequences are "unwound" to their original type until a simple IDL type or user-defined IDL type (of the non `typedef` variety) is encountered. Typedefs for arrays and sequences are converted to corresponding named classes. Thus the IDL `typedef`

`sequence<octet> Bitmap` is converted to a corresponding Bitmap class. An example of a Bitmap class is given below:

```
// Java
package org.havi.types;
public final class Bitmap extends HaviObject {
      private byte[] value;

      public Bitmap() {…}

      public Bitmap(byte[] value) throws HaviInvalidValueException {
            …
            this.value = value;
            …
      }

      public Bitmap(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException {
            …
            this.unmarshal(hbais);
            …
      }

     public byte[] getValue() {
            return(value.clone());
      }

      public void setValue(byte[] value) throws HaviInvalidValueException {
            this.value = value;
      }

      public Object clone() {…}
      public boolean equals(Object o) {…}
      public int hashCode() {…}
      public void marshal(HaviByteArrayOutputStream hbaos) throws HaviMarshallingException {…}
      public void unmarshal(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException {…}
}
```

In addition all classes defined for `typedef` must extend either the abstract HaviObject class or the abstract HaviImmutableObject class. They must implement the abstract methods equals, hashCode, marshal and clone. In addition, the `unmarshal` method must be implemented if the classes defined for typedef extends HaviObject. All the classes must provide a constructor for constructing an instance of the object from a HaviByteArrayInputStream. This is further explained in the Marshalling and Unmarshalling section.

7.3.5 Holder Classes

Support for `out` and `inout` parameter passing modes requires the use of additional “holder” classes. These classes are available for all the basic IDL datatypes and sequences. For user defined IDL types, such as `struct` and `union`, if the types are mutable, no additional holder classes are required. For user defined IDL types, such as `struct` and `union`, if the types are immutable, the holder class name is the Java type name (with its initial letter capitalized) to which the datatype is mapped with an appended Holder. For the basic IDL datatypes, the holder class name is the Java type name (with its initial letter capitalized) to which the datatype is mapped with an appended Holder. (E.g. IntHolder for an `int`.) For `sequence` the holder class is defined by using the same name as the type of the sequence with SeqHolder appended to it.

Each holder class must extend from the HaviHolder class. Each holder class has a pair of the accessor method named getValue and the modifier method named setValue for an internal private field named "value" which holds a basic data value, an array instance or an immutable instance. Each holder class provides a constructor to initialize the value field of a holder instance, and additionally provides a default constructor (the value field can be updated after the object creation). The default

constructor sets the value field to the default value for the type of the value field as defined by the Java language: FALSE for boolean, 0 for numeric and char types and null for arrays and other classes. Each holder class must implement equals, hashCode, marshal and unmarshal methods. In addition each holder class must provide a constructor for constructing a holder instance from a HaviByteArrayInputStream. This is further explained in section 7.3.7 on marshalling and unmarshalling. The behavior of the accessor method getValue(), the modifier method setValue(), and the constructor initializing the value field shall be as follows:

- If the value field is a subclass of the HaviImmutableObject class, a reference to the object is got/set.
- If the value field is an array class, a reference to the object is got/set.
- If the value field is of the basic data type, the basic data is got/set by value

An example of a holder class for a basic type and a sequence type is given below:

```
// Java
package org.havi.types;
public final class ShortHolder extends HaviHolder {
    private short value;

    public ShortHolder() { …}

    public ShortHolder(short value) throws HaviInvalidValueException {
        …
        this.value = value;
        …
    }

    public ShortHolder(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException {
        …
        this.unmarshal(hbais);
        …
    }

    public short getValue() {
         return(value);
    }

    public void setValue(short value) throws HaviInvalidValueException {
         this.value = value;
    }

    public boolean equals(Object o) { … }
    public int hashCode() { … }
    public void marshal(HaviByteArrayOutputStream hbaos)throws HaviMarshallingException {  … }
    public void unmarshal(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException { … }

}


// Java
package org.havi.types;
public final class SeidSeqHolder extends HaviHolder {
    private SEID[] value;

    public SeidSeqHolder() { …}

    public SeidSeqHolder(SEID[] value) throws HaviInvalidValueException {
        …
        this.value = value;
        …
```

```
    }

    public SeidSeqHolder (HaviByteArrayInputStream hbais) throws HaviUnmarshallingException {
        …
        this.unmarshal(hbais);
        …
    }

    public SEID[] getValue() {
         return(value.clone());
    }

    public void setValue(SEID[] value) throws HaviInvalidValueException {
        this.value = value;
    }

    public boolean equals(Object o) { …}
    public int hashCode() { … }
    public void marshal(HaviByteArrayOutputStream hbaos) throws HaviMarshallingException { … }
    public void unmarshal(HaviByteArrayInputStream hbais) throws HaviUnmarshallingException { … }
}
```

7.3.6 Exceptions

The IDL exceptions are not supported. However, the HAVi Java APIs define a `HaviException` corresponding to each error code and in addition define a number of other exceptions. All exceptions corresponding to HAVi error codes in HAVi Java APIs are extended from the base class `HaviException`. These classes are described in detail in Appendix A. Refer to section 7.3.8.3 on Error Codes to note when exceptions are raised.

7.3.6.1 Exception Throwing and Handling

The following section applies to all packages in the org.havi package namespace with the exception of org.havi.ui and sub-packages of that package. Exceptions to be thrown in HJA are divided into the following two categories: Subclasses of the HaviException class and Subclasses of the Exception class.

Exceptions which are the subclasses of the HaviException class correspond to the return error codes of HAVi API (See Section 7.3.8.3.). An exception in this category is thrown if an error code was received from another software element or if an error code should be returned to another software element. And since a client shall interpret a received unknown error code as `EUNIDENTIFIED_FAILURE`, HaviUnidentifiedFailureException shall be thrown if an unknown error code is received. In addition, Java clients shall catch any HaviException to avoid termination on an unknown HaviException thrown by a `HaviClient` of a newer FAV (See Section 5.1.7.).

The subclasses of the Exception class are for notifying of abnormalities or exceptional events among the instances of the classes defined in HJA. There are the following six exceptions in this category:

- HaviInvalidValueException
- HaviMarshallingException
- HaviUnmarshallingException
- HaviUnionException
- HaviMsgListenerExistsException
- HaviMsgListenerNotFoundException

On execution of a constructor or a modifier method in a subclass of the HaviObject class, HaviInvalidValueException is thrown if at least one argument is invalid.

On execution of any constructor or any method other than constructors and modifier methods in subclasses of the HaviObject class, java.lang.IllegalArgumentException (and not HaviInvalidValueException) is thrown if at least one argument is invalid.

On execution of any constructor or any method in HJA, unless the HAVi Specification or the Javadoc explicitly states that null can be used in the constructor or method, java.lang.NullPointerException (and not HaviInvalidValueException) is thrown if at least one argument is null (regardless of whether the class which provides these constructor or method extends the HaviObject class or not.).

If HaviInvalidValueException is caught on execution of a constructor or a method in subclasses of the HaviClient class, HaviServerHelper class, or HaviListener class, the constructor or method shall handle this exception in one of the following ways:

- solve the exceptional event within the method itself
- throw the exception HaviInvalidParameterException
- send the error code EINVALID_PARAMETER to an appropriate software element

On execution of a marshal method in a subclass of the HaviObject or HaviHolder class HaviMarshallingException is thrown if marshalling fails for some reason. Some possible reasons could be, use of invalid data (e.g. marshalling a null instance) or the lack of resources (e.g. marshalling extremely long array data).

On execution of an unmarshal method or an unmarshal constructor in a subclass of the HaviObject or HaviHolder class HaviUnmarshallingException is thrown if unmarshalling fails for some reason. Some possible reasons could be, use of invalid data (e.g., the message data is shorter than expected or the first value of a sequence is negative) or the lack of resources (e.g., unmarshalling extremely long array data).

If HaviMarshallingException or HaviUnmarshallingException is caught, it is impossible to determine whether the cause of this exception is the use of invalid data or an abnormality in execution environments. Thus, the constructor or method which caught HaviMarshallingException or HaviUnmarshallingException shall handle this exception in one of the following ways:

- solve the exceptional event by the method itself
- throw the exception HaviUnidentifiedFailureException
- send the error code EUNIDENTIFIED_FAILURE to an appropriate software element

On execution of an accessor method in a subclass of the HaviObject to which union is mapped, HaviUnionException is thrown if the discriminator value is different from the value corresponding to the accessor method. For example, EventId.getSystemEid() throws HaviUnionException if this method is executed when EventId is not a system event (See Section 7.3.4.3).

On execution of the addHaviListener method in the SoftwareElement class, HaviMsgListenerExistsException is thrown if the instance of subclasses of the HaviListener class specified as argument has already been attached to the instance of the SoftwareElement class.

On execution of the removeHaviListener method in the SoftwareElement class, HaviMsgListenerNotFoundException is thrown if the instance of subclasses of the HaviListener class specified as argument has not been attached to the instance of the SoftwareElement class.

7.3.6.2 States of Instance and Arguments

When an exception is thrown from a method of an instance, the state of the instance and the state of each argument of the method shall be defined by the following rules:

1) For any classes in HJA:

1-1) When any exception is thrown by any constructor or method and unless the HAVi Specification or the Javadoc explicitly describes the states of the instance or the arguments:

- The state of the instance which threw the exception is indeterminate.
- The state of each argument of the method is indeterminate.

2) For subclasses of the HaviObject class, subclasses of the HaviHolder class, the HaviByteArrayOutputStream class, and the HaviByteArrayInputStream class:

2-1) When a HaviUnionException is thrown by any method:

- The state of the instance which threw the exception does not change.

2-2) When a HaviInvalidValueException is thrown by any constructor or method:

- The state of the instance which threw the exception does not change.
- The state of each argument of the method does not change.

2-3) When a HaviMarshallingException is thrown by any method:

- The state of the instance which threw the exception does not change.

3) For the SoftwareElement class:

3-1) When a HaviMsgListenerExistsException, a HaviMsgListenerNotFoundException, or any exception which is of a subclass of the HaviException class is thrown by any method:

- The state of the instance which threw the exception does not change.
- The set of the attached listeners of the instance which threw the exception does not change.
- Each state of the attached listeners of the instance which threw the exception does not change.

3-2) When any exception other than a HaviMsgListenerExistsException, a HaviMsgListenerNotFoundException, or an exception which is of a subclass of the HaviException class is thrown by any method:

- The set of the attached listeners of the instance which threw the exception does not change.
- Each state of the attached listeners of the instance which threw the exception does not change.

4) For the subclasses of the HaviClient class and the HaviServerHelper class:

4-1) When any exception which is of a subclass of the HaviException class is thrown by any message-sending method:

- The state of the instance which threw the exception does not change.

7.3.7 Marshalling and Unmarshalling

HAVi requires that messages are sent and received following the Common Data Representation standard (see section 3.2.3.4). To facilitate sending of Java objects around the HAVi network, all data type classes used in the HAVi Java APIs, including “holder” classes, shall implement “marshalling” of the objects into output data stream and also implement “unmarshalling” of the input data. Every class for a type shall implement the marshal and unmarshal methods. In cases of classes extending from HaviImmutableObject class, the unmarshal method shall just throw HaviUnmarshallingException. All classes shall also provide a constructor, which accepts an input data stream and constructs itself by reading it. When marshalling bounded arrays and sequences, the marshal method shall raise a HaviMarshallingException if the size of the array or the sequence exceeds

the specified limit. Similarly while unmarshalling HaviUnmarshallingException shall be raised if the incoming stream has more data then the specified limit. HaviUnmarshallingException shall be also raised when a value of unmarshalled data is invalid with respect to the class.

7.3.7.1 HaviByteArrayInputStream & HaviByteArrayOutputStream

HAVi Java APIs provides two stream classes that provide methods for “marshalling” and “unmarshalling” messages.

HaviByteArrayInputStream class extends the java.io.ByteArrayInputStream class and provides for reading incoming data that is in Common Data Representation (CDR) format as specified in section 3.2.3.4.

HaviByteArrayOutputStream class extends the java.io.ByteArrayOutputStream class and provides for writing out data that is in CDR format.

7.3.8 HaviClient and HaviServerHelper

7.3.8.1 Client and Server

In the HAVi specification, several IDL interfaces are defined. Each interface defines several operations. For example, `Cmm1394::GetGuidList`, where `Cmm1394` is the IDL interface and `GetGuidList` is the operation. Each IDL interface is basically mapped to a Java client class and a server helper class. Client classes and server helper classes are explained below.

The HAVi Java APIs can be used to implement client objects (software elements which would like to reach a server’s software element API, as havlets, Application Modules, DCMs, FCMs, or system software elements) and server objects (DCMs or FCMs for example).

7.3.8.1.1 Client Classes

To implement client objects, the HAVi Java APIs provide “client classes”.

There are two kinds of client classes. The first kind are called “remote client classes” or simply “client classes”. The second kind are called “local client classes”. (Some APIs have only “remote client classes” because such APIs does not distinguish remote access and local access.)

Remote client classes provide methods to access software elements of a specified type. The methods construct appropriate messages and send them to the specified software elements. In the case of asynchronous methods the result is returned to all listeners installed by the caller. In the case of synchronous methods the resulting byte stream is parsed back and returned to the caller. All remote client classes are extended from the `HaviClient` class.

Local client classes provide methods to access local software elements of a specified type. In the case of asynchronous methods the result is returned to all listeners installed by the caller. In the case of synchronous methods the resulting byte stream is parsed back and returned to the caller. All local client classes are extended from the corresponding remote client class.

For example, the remote client class of the `Cmm1394` API is defined as follows:

```
// Java

public class Cmm1394Client extends HaviClient {
    …
```

```
}
```

And the local client class of the Cmm1394 API is defined as follows:

```
// Java

public class Cmm1394LocalClient extends Cmm1394Client {
    …
}
```

In the client classes, each IDL operation is mapped to two Java methods: one with the same name (except starting in lowercase), the other one with the same name (again starting in lowercase) suffixed with Sync. For example:

```
//IDL
Status Cmm1394::GetGuidList(
      out sequence<GUID> activeGuidList,
      out sequence<GUID> nonactiveGuidList)

//Java

public class Cmm1394LocalClient extends Cmm1394Client {
    …

    public final void getGuidList(IntHolder transactionId)
        throws HaviGeneralException,
            HaviMsgException { ... }

    public final void getGuidListSync(int timeout,
            GuidSeqHolder activeGuidList,
            GuidSeqHolder nonactiveGuidList)
        throws HaviGeneralException,
            HaviMsgException,
            HaviCmm1394NotReadyException { ... }
    …
}
```

The first method, getGuidList, will send the message that corresponds to the IDL Cmm1394::GetGuidList API using the Messaging System's MsgSendRequest API. The method does not wait for, or handle, the reply message coming back. The result is returned to all listeners installed by the caller. Therefore, the signature only has the in and inout parameter of the IDL definition. Moreover, it returns the transactionId used by the Messaging System in sending the message; this allows the caller to match the corresponding incoming reply later on.

The second method, getGuidListSync, sends the call via the Messaging System's MsgSendRequestSync API. The method waits for the reply message and handles the reply by providing the result in the out and inout parameters. Therefore, the signature provides all in, out and inout parameters. Moreover, the signature provides a parameter to specify the timeout to be used in the MsgSendRequestSync API. A value of "0" for timeout would result in the default timeout of the Messaging System.

In case of a local API such as the above case, note that this message transfer is merely a logical model and whether MsgSendRequest or MsgSendRequestSync API is actually called and the corresponding message is actually sent is implementation dependent.

Client classes only contain the methods accessible by non-system components, i.e. methods that may be accessed from any uploaded entity. They do not contain methods that can only be called by a system component and which may be handled in a proprietary way.

7.3.8.1.2 Server Helper Classes

To help the designer of a non-system software elements (e.g., DCMs and FCMs) the HAVi Java APIs provide “server helper classes”. These classes contain methods to send back responses to incoming requests and also methods to call message back APIs (see section 5.1.1). All server helper classes are extended from the HaviServerHelper class.

A response method gets as parameters a destination SEID, a return code, a transaction ID (as defined in `MsgSendResponse`) and all output parameters defined in the specific IDL API. The instance of the SoftwareElement class which provides the `MsgSendResponse` API and the transfer mode of the `MsgSendResponse` API are given through the constructor of the server helper class. The SoftwareElement class is described in section 7.3.9.

The following example is the server helper class for a VCR FCM. The response method corresponding to the API `Vcr::Play` is described. The instance of the SoftwareElement class and the transfer mode are given through the constructor of the VcrServerHelper class.

```
// Java

public class VcrServerHelper extends HaviServerHelper {

    public VcrServerHelper(SoftwareElement se,
        int transferMode) { ... }

    public final void playResp(SEID destSeid,
            Status returnCode,
            int transactionId)
        throws HaviGeneralException,
            HaviMsgException { ... }
    …
}
```

The following example is the server helper class for an FCM. The response method corresponding to the `Fcm::SubscribeNotification` API is described. The two methods corresponding to the message back for the `<Client>::FcmNotification` (which is implemented in the client) are also described.

```
// Java

public class FcmServerHelper extends HaviServerHelper {

    public FcmServerHelper(SoftwareElement se,
        int transferMode) { ... }

    public final void subscribeNotificationResp(SEID destSeid,
            Status returnCode,
            int transactionId,
            HaviByteArrayOutputStream currentValue,
            short notificationId)
        throws HaviGeneralException,
            HaviMsgException { ... }

    …

    void fcmNotification(
            SEID destId,
            IntHolder transactionId,
            OperationCode opCode,
            short notificationId,
            short attributeIndicator,
            HaviByteArrayOutputStream value)
        throws HaviGeneralException,
```

```
                HaviMsgException { ... }

    void fcmNotificationSync(
                SEID destId,
                int timeOut,
                OperationCode opCode,
                short notificationId,
                short attributeIndicator,
                HaviByteArrayOutputStream value)
            throws HaviGeneralException,
                HaviMsgException { ... }

    ...
}
```

7.3.8.2 Parameter Passing Modes

IDL `in` parameters, which implement call-by-value semantics, are mapped to normal Java actual parameters. If an operation has a return value other than `SUCCESS` in a HAVi message, then a corresponding HaviXXXException is raised in the code of the operation caller (usually the client-side, but the server-side in the case of a MB, or "message back", operation). Note that an exception may be thrown at either the client-side or the server-side, but never on both sides. `in` parameters are not modified by the procedure.

IDL `out` and `inout` parameters, which implement call-by-result and call-by-value/result semantics, cannot be mapped directly into the Java parameter passing mechanism for basic types and sequences. This mapping defines additional holder classes for all the IDL basic and sequence types, which are used to implement these parameter modes in Java. The client supplies an instance of the appropriate Java holder class that is passed (by value) for each IDL `out` or `inout` parameter. The contents of the holder instance (but not the instance itself) are modified by the invocation, and the client uses the (possibly) changed contents after the invocation returns. In the case of user-defined types such as `struct` and `union`**,** there are no additional holder classes needed and the client supplies an instance of the appropriate Java class itself.

Parameters defined as `out` or `inout` in the IDL description may be modified by the procedure. This concerns the object passed as parameters as well as all (sub) objects to which the parameter refers. This allows the caller to allocate and reuse the classes needed for the reply and alleviates the need for creating new objects by the method on each call.

All types of variables (`in`, `out` and `inout`) shall not be changed by the environment (in a separate thread) during the complete duration of the call.

```
// IDL
module Example {
   interface Modes {
      void operation (in long inArg,
                      out long outArg,
                      inout long inoutArg);
   }
}
```

```
// Java
package Example;
public interface Modes {
    void operation(int inArg,
                   IntHolder outArg,
                   IntHolder inoutArg);
    }
```

```
}
```

In the above the actual `in` parameter comes in as only an ordinary value. But for the `out` and `inout` parameters, an appropriate holder, if necessary, must be constructed. A typical use case might look as follows:

```
// use Java code
// select a target object
Example.Modes target = …;

// get the in actual value
int inArg = 57;

// prepare to receive out
IntHolder outHolder = new IntHolder();

// set up the in side of the inout
IntHolder inoutHolder = new IntHolder(131);

// make the invocation
int result = target.operation(inArg, outHolder, inoutHolder);

// use the value of the outHolder
… outHolder.setValue() …

// use the value of the inoutHolder
… inoutHolder.getValue() …
… inoutHolder.setValue() …
```

Before the invocation, the input value of the `inout` parameter must be set in the holder instance that will be the actual parameter. The `inout` holder can be filled in either by constructing a new holder from a value or by assigning to the value of an existing holder of the appropriate type. After the invocation, the client uses the outHolder.setValue() to set the value of the `out` parameter, and the inoutHolder.getValue() to access the output value of the `inout` parameter. The return result of the IDL operation is available as the result of the invocation.

7.3.8.3 Error Codes

The return error codes in the HAVi specification are mapped to exceptions of similar name prefixed with HAVi, suffixed with Exception and the beginning "E" removed and "_" removed. Each word is started with uppercase and the remaining letters are in lowercase. Thus the error code `ENOT_READY` is translated to a HaviNotReadyException class. If the error code belongs to a particular API then the API name is inserted immediately after the prefix HAVi. Thus `Msg::ENOT_READY` would translate to HaviMsgNotReadyException. The HAVi Java APIs define in addition several other exceptions. These exception classes can be found in the org.havi.types package.

Note that it might be the case that an exception is raised but that the `out` and `inout` parameters still have proper values.

7.3.8.4 Parameter Checking

7.3.8.4.1 Parameter Checking Before Sending Messages

Before any message is sent, all parameters in the message shall be checked as follows:

- On execution of a constructor or a modifier method in a subclass of the HaviObject class or a subclass of the HaviImmutableObject class:
  - when a value which is not allowed by the HAVi Specification is attempted to be set to

any "basic data type" field of the instance, a HaviInvalidValueException shall be thrown.
- when null is attempted to be set to any “class” field of the instance, a java.lang.NullPointerException shall be thrown.

- On execution of a message-sending method in the SoftwareElement class, a subclass of the HaviClient class, or a subclass of the HaviServerHelper class:
  - when a value which is not allowed by the HAVi Specification is specified as any "basic data type" "in" parameter of the method, a HaviInvalidParameterException shall be thrown.
  - when a holder instance which is specified as any "basic data type holder class", "out/inout" parameter of the method turns out to hold a value which is not allowed by the HAVi Specification, a HaviInvalidParameterException shall be thrown.
  - when a holder instance which is specified as any "basic data type sequence holder class", "out/inout" parameter of the method turns out to hold a value which is not allowed by the HAVi Specification, a HaviInvalidParameterException shall be thrown.
  - when null is specified as any "class" parameter of the method, a java.lang.NullPointerException shall be thrown.

7.3.8.4.2 Parameter Checking After Receiving Message

After any message is received, all parameters in the message shall be checked as follows:

- On execution of an unmarshall constructor in any classes:
  - when a value which is not allowed by the HAVi Specification is attempted to be set to any "basic data type" field of the instance, a HaviUnmarshallingException shall be thrown.

- On execution of a synchronous message-sending method in the SoftwareElement class, a subclass of the HaviClient class, or a subclass of the HaviServerHelper class:
  - when a value which is not allowed by the HAVi Specification is set as the held value of any "basic data type holder class", "out/inout" parameter of the method, a HaviInvalidParameterException shall be thrown.
  - when a value which is not allowed by the HAVi Specification is set as at least one of the held values of any "basic data type sequence holder class", "out/inout" parameter of the method, a HaviInvalidParameterException shall be thrown.

7.3.9 SoftwareElement

The SoftwareElement class implements safe access to the HAVi Messaging System for a single software element. A fresh SEID is created for a software element during construction of a SoftwareElement object. To send messages, class SoftwareElement implements the Messaging System send API primitives, without the SEID `sourceSeid` parameter. The SEID of the source (or sender) of a message is implicitly contained in the SoftwareElement object.

To handle incoming messages, HAVi defines the notion of a HAVi listener. All HAVi listener classes extend the HaviListener class and implement one abstract method called receiveMsg. This receiveMsg method generally filters and processes a message, received from the underlying system (the message dispatcher).

A listener is installed through the addHaviListener methods of the SoftwareElement class or via the SoftwareElement(HaviListener) constructor.. A listener can specify reception of all incoming messages for the software element, or only messages originating from a particular sender SEID.

Received messages are delivered to all HaviListener objects installed for the associated

SoftwareElement object (and optionally sender SEID).

Received messages are delivered to all HaviListener objects installed for the associated SoftwareElement object (and optionally sender SEID). The receiveMsg method of the listeners are invoked in the order that the listeners have been installed.

For a given message received by a SoftwareElement object, the haveReplied argument of the listeners' receiveMsg method shall be set to false until a listener has returned true. After a listener has returned true, all following listeners' receiveMsg shall have haveReplied set to true.

In the case that the message is a request, and all listeners have returned false, the SoftwareElement shall send a response with the error code UNKNOWN_MESSAGE. If one or more listeners return true the SoftwareElement shall not send a response.

The implementor of the listener should be aware that during the time that the receiveMsg method blocks, the related SoftwareElement may not be able to process other incoming messages.

How messages incoming to a device are dispatched to the relevant SoftwareElement objects is implementation dependent. Also, whether and how a receiving Messaging System implements the noack (`Msg::ETARGET_REJECT`) message/mechanism is SoftwareElement (or Messaging System) implementation dependent.

A SoftwareElement object has its own thread to call the HaviListener.receiveMsg callbacks; it does not block other SoftwareElement objects or the underlying Messaging System while performing these callbacks. Consequently, there are no restrictions on the implementations of the (application provided) HaviListener.receiveMsg method: the method does not have to be treated as an interrupt. However, one must be careful when using SoftwareElement.msgSendRequestSync in the HaviListener.receiveMsg method, since this will block the calling software element until the response to the supplied request is received. While the msgSendRequestSync is blocked no other incoming messages can be processed by the software element. This may result in a deadlock.

For example, suppose that a registry is implemented in such a way that it uses msgSendRequestSync within its HaviListener.receiveMsg to forward `GetElement` requests to other registries. When two applications on two different nodes (*A1* and *A2*, respectively) happen to query their registries (*R1* and *R2*, respectively) at approximately the same time, a deadlock may occur. *R1* is busy handling the request of *A1* and waits for the response to the `GetElement` forwarded to *R2*. During this period, *R1* cannot process incoming messages. Furthermore, *R2* is busy handling the request of *A2* and waits for the response to the `GetElement` forwarded to *R1*. Also, during this period, *R2* cannot process incoming messages. This is a deadlock situation which will result in timeouts of the msgSendRequestSync call in both *R1* and *R2*.

A general solution is to avoid synchronous sends in cases where it might lead to deadlock, or to add sufficient threads in the Messaging System clients. For example, to avoid the scenario described above, one additional thread per Registry would suffice.

Java programs will access the class SoftwareElement in a multi-threaded fashion. To avoid unnecessary EBUSY exceptions it is preferable that the implementation of the class SoftwareElement either supports outstanding messages as defined in chapter 3.2.1.2.8 or serializes any parallel messages.

7.4 Code Units

HAVi Java code units are entities for uploading Java bytecode by FAV nodes. Basically, the format of a Java code unit is the Java archive or “JAR” format as specified in Java 1.1 [7][8]. The class

loader of an FAV resolves class and resource names relative to the root directory of the archive. For the different type of code units, i.e., DCM code units, Application Module code units and havlet code units, there are different requirements on the actual class definitions that must be in the JAR file. The way they are handled by an FAV is basically the same:

- The FAV loads the classes in code unit JAR file such that they are available for the code unit. Classes in package <none> are located in the root directory of the archive.
- The FAV finds a specific class definition with the predefined name (the different types of code units are specified below) and makes a new instance of the specified class.
- The FAV calls the install method of the newly created object with proper parameters for that type of class. Also, the FAV provides a reference to an UninstallationListener interface (see below) by which the installed object can indicate when it has uninstalled itself.
- When the installed object indicates its installation via an UninstallationListener, the FAV releases the reference to the object to allow the removal via the Java garbage collector.

To allow indication of uninstallation, the FAV provides an UninstallationListener which has the following interface (in package org.havi.system):

```
public interface UninstallationListener{

    /*
     * To be called by the installed object to indicate
     * that it has removed itself completely (removed
     * all subscriptions, unregistered and closed its
     * message handle) and that the reference to the
     * object can be removed and the object can be
     * destroyed by the garbage collector.
     */
    public void uninstalled();
}
```

7.4.1 DCM Code Units

A DCM code unit is a code unit consisting of code to install a DCM. As for all code units, the format is a Java JAR file. Specific for DCM code units is that they must contain a concrete class named DcmCodeUnit in package <none> that implements DcmCodeUnitInterface:

```
public class DcmCodeUnit
    implements org.havi.system.DcmCodeUnitInterface {
    …
}
```

This allows the DCM manager to find the proper class for installation of the DCM code unit.

The interface DcmCodeUnitInterface is an interface defined in org.havi.system as:

```
public interface DcmCodeUnitInterface {

   public int install(GUID  nodeId, UninstallationListener listener );

   public void uninstall();
}
```

DcmCodeUnit::install

Prototype

```
public int install(
```

```
        GUID nodeId,
        UninstallationListener listener);
```

Parameters

- nodeId the GUID referring to the device the DCM corresponds to
- listener reference to a listener to be called on uninstallation

Description

Installs a DCM code unit. The GUID provides means to communicate with a guest device via the CMM. The install() method shall install exactly one DCM and zero or more FCMs associated with the DCM. The uninstalled() method of the provided listener is invoked by the DCM code unit to notify its installer (the local DCM Manager) that all its software elements have unsubscribed their subscriptions, have been unregistered, have closed their messages handles and released access to the corresponding guest device. This allows removal of the object by the garbage collector and a new DCM code unit to be installed for the guest device, if appropriate.

The install method should only take care of installation and should return as soon as possible (it shall not provide a thread for execution).

Return value

- 0: if the DCM code unit has been successfully installed.
- 1: if the installation failed and no DCM software elements have been installed.

DcmCodeUnit::uninstall

Prototype

```
public void uninstall();
```

Description

This method makes the DCM code unit abort its activities at once. The code unit takes care that all its software elements have unsubscribed their subscriptions, have been unregistered, have closed their messages handles and have released access to the corresponding guest device. It indicates its uninstallation via the uninstalled() method of the provided listener.

7.4.2 Application Module Code Units

An Application Module code unit is a code unit consisting of code to install an Application Module. As for all code units, the format is a Java JAR file. Specific for Application Module code units is that they must contain a concrete class named AMCodeUnit in package <none> that implements AMCodeUnitInterface:

```
public class AMCodeUnit
    implements org.havi.system.AMCodeUnitInterface {
    …
}
```

This allows an FAV to find the proper class for installation of the Application Module's code unit.

The interface AMCodeUnitInterface is an interface defined in org.havi.system as:

```
public interface AMCodeUnitInterface {

  public int install(
    TargetId              targetId,
    boolean               n1Uniqueness,
    UninstallationListener listener );
```

```
    public void uninstall();
}
```

AMCodeUnit::install

Prototype

```
public int install(
        TargetId  targetId,
        boolean n1Uniqueness,
        UninstallationListener listener );
```

Parameters

- targetId Target ID of the Application Module
- n1Uniqueness indication of whether the n1 field in targetId has been assigned so as to be persistently unique to this application
- listener reference to a listener to be called on uninstallation

Description

Installs an Application Module code unit The install() method shall install exactly one AM. The Target ID of this Application module is the Target ID provided by the caller of this method (the host on which this Application Module is installed) in the targetId parameter. The targetID has been constructed by the host according to the description given in the Target ID definition in section 5.6.2.

The Application Module constructs its HAVi Unique ID based on values of the parameters provided by the host. The targetId and the n1Uniqueness field of the HUID of this Application Module shall be the same as the targetId and n1Uniqueness field provided by the caller, the other fields of the HUID may be decided by this Application Module itself (according to the rules for HUIDs).

The unInstalled() method of the provided listener is invoked by the Application Module code unit to notify to its installer that all its software elements have unsubscribed their subscriptions, have been unregistered and have closed their messages handles. This allows removal of the object by the garbage collector.

The install method should only take care of installation and should return as soon as possible (it shall not provide a thread for execution).

Return value

- 0: if the Application Module code unit has been successfully installed.
- 1: if the installation failed and no Application Module software elements have been installed.

AMCodeUnit::uninstall

Prototype

```
public void uninstall();
```

Description

This method makes the Application Module code unit abort its activities at once. The code unit takes care that all its software elements have unsubscribed their subscriptions, have been unregistered and have closed their messages handles. It indicates its uninstallation via the uninstalled() method of the provided listener.

7.4.3 Havlet Code Units

A havlet code unit is a code unit consisting of Java bytecode to install a havlet. As for all Java code units, the format is a Java JAR file. Specific for havlet code units is that they must contain a concrete class named HavletCodeUnit in package <none> that implements HavletCodeUnitInterface:

```
public class HavletCodeUnit
      implements org.havi.system.HavletCodeUnitInterface {
      …
}
```

This allows an FAV to find the proper class for installation of the havlet code unit.

The interface HavletCodeUnitInterface is an interface defined in org.havi.system as:

```
public interface HavletCodeUnitInterface {

   public int install(
      SEID   source,
      UninstallationListener listener );

   public void uninstall();
}
```

HavletCodeUnit::install

Prototype

```
public int install(
      SEID  source,
      UninstallationListener listener );
```

Parameters

- source SEID of the source where the code unit has been uploaded from
- listener reference to a listener to be called on uninstallation

Description

Install the havlet code unit. source must be the SEID of the source (DCM or Application Module) from which the havlet code unit has been retrieved. This provides the havlet with a communication mechanism to its source, via standard HAVi messaging. The install() method shall install exactly one havlet. The uninstalled() method of the provided listener is invoked by the havlet code unit to notify to its installer that all its software elements have unsubscribed their subscriptions, have been unregistered and have closed their messages handles This allows removal of the object by the garbage collector.

Return value

- 0: the havlet code unit has been successfully installed.
- 1: no havlet code unit has been installed.

HavletCodeUnit::uninstall

Prototype

```
public void uninstall();
```

Description

This method makes the havlet code unit abort its activities at once. The code unit takes care that all its software elements have unsubscribed their subscriptions, have been unregistered and have

closed their messages handles. It indicates its uninstallation via the uninstalled() method of the provided listener.

7.5 Isochronous Data Processing

Virtual FCMs, introduced in section 3.5.2.5, allow the construction of HAVi applications that process isochronous data. This section describes the APIs provided by HAVi for such applications.

Implementation of an FCM in general depends on whether the FCM is embedded (implemented in native code) or uploaded (implemented in Java bytecode). HAVi does not specify native APIs, so implementation of an embedded FCM (whether physical or virtual) is entirely platform dependent (however the embedded FCM must respond to the HAVi messages specified in section 5.7.3). In the case of uploaded FCMs, which are platform independent, HAVi must specify the Java APIs needed for their implementation. The org.havi.system package provides interfaces to the main HAVi system components: the Messaging System, Communication Media Manager, Event Manager and Registry. This package is sufficient for developing an uploaded physical FCM. Uploaded virtual FCMs, on the other hand, require additional support. The org.havi.system package provides sufficient functionality for implementing a control interface (note, this includes controlling the content interface) but does not support implementation of the content interface itself.

For example, consider a virtual FCM which sinks isochronous data. The Stream Manager can be used to establish a connection, resulting in data flowing to the FCM, but neither the Stream Manager nor other HAVi system components include APIs which expose isochronous data. In order to process isochronous data, virtual FCMs must use the org.havi.Iec61883 package.

The org.havi.Iec61883 package supports the IEC 61883.1 protocol for transmission of isochronous data over IEEE 1394. It consists of the following classes:

Iec61883InputStream
Iec61883OutputStream

Iec61883InputStream

public class Iec61883InputStream
extends java.io.InputStream

Iec61883InputStream allows virtual FCMs to consume data from a 1394 isochronous channel. The read() method provides the virtual FCM with the payload data from IEC 61883.1 CIP packets. Depacketization of CIP packets is handled by Iec61883InputStream.

Iec61883OutputStream

public class Iec61883OutputStream
extends java.io.OutputStream

Iec61883OutputStream allows virtual FCMs to produce data on a 1394 isochronous channel. The write() method provides the payload data for IEC 61883.1 CIP packets. Packetization and transmission timing are handled by Iec61883OutputStream.

7.5.1 An Example

The following example illustrates how the above classes could be used in the implementation of a virtual FCM. This example is not complete and is merely intended to illustrate how the org.havi.Iec61883 classes relate to each other and to give a possible template for their use.

```
import org.havi.constants;
import org.havi.types;
```

```
import org.havi.system;
import org.havi.Iec61883;

class ExampleVirtualFCM extends FcmServerHelper {
    // this virtual FCM has N input plugs and M output plugs
    private Iec61883InputStream[]  src = new Iec61883InputStream[N];
    private Iec61883OutputStream[] sink = Iec61883OutputStream[M];

    public void IecAttachResp(SEID destSeid, Status returnCode, int transactionId,
        IecPlug pcr, InternalPlug plug)
    {
        // check for error conditions, if ok make stream
        // first find channel, assume myDcm is  DcmClient for parent DCM
        //
        myDcm.getChannelUsageSync(new IntHolder tid, pcr, new ShortHolder channel);
        if(pcr.getDir() == ConstDirection.IN)
            src[plug.getPlugNum()] = new Iec61883InputStream(channel.getValue());
        else
            sink[plug.getPlugNum()] = new Iec61883OutputStream(channel.getValue());
    }
    public void iecDetachResp(SEID destSeid, Status returnCode, int transactionId,
        IecPlug pcr, InternalPlug plug)
    {
        // check for error conditions, if ok destroy stream
        if(pcr.getDir() == ConstDirection.IN)
            src[plug.getPlugNum()] = null;
        else
            sink[plug.getPlugNum() = null;
    }
}
```

7.5.2 Relationship with the Stream Manager

In the above example, the “plug control register” to be processed by ExampleVirtualFcm, is passed as a parameter in iecAttachResp. Typically this parameter is not selected by the application but rather by the Stream Manager. The DCM is then informed of this selection via `Dcm::Connect`. A virtual FCM should be handled, by the Stream Manager, in the same manner as a physical FCM. Thus the recommended implementation for virtual FCMs is that `Dcm::Connect` invoke `Fcm::IecAttach` with the plug control register selected by the Stream Manager. Processing this request by the FCM will result in the invocation of iecAttachResp.

7.6 Example: A DCM Code Unit and DCM (Informative)

This section gives an example of the use of the HAVi Java APIs to implement a bytecode DCM and its DCM code unit. This example is provided for informative purposes and is not intended as an implementation blueprint. The example is not a full DCM implementation, but does demonstrate use of the HAVi Messaging System and Registry by a bytecode DCM.

The example consists of the following classes:

- MyDcm – implements the DCM APIs
- MyDcmListener – the HaviListener used by MyDcm
- DcmCore – dispatches HAVi messages for MyDcm
- MyDcmCodeUnit – the code unit used to install an instance of MyDcm

7.6.1 MyDcm.java

```
// package com.someone.dcm.test;

import org.havi.types.*;
import org.havi.constants.*;
import org.havi.system.*;

public class MyDcm {
	// fields
	//
	public HUID dcmId;
	public GUID nodeId;

	public MyDcmListener listener;
	public SoftwareElement mySe;
	public SEID mySeid;

	private Cmm1394LocalClient cmm;
	private RegistryLocalClient registry;

	private int deviceClass = ConstDeviceClass.BAV;
	private String manufacturer = “Sample”;
	private String userName = “SampleName”;

	// constructor
	//
	public MyDcm(HUID _dcmId, GUID _nodeId) {
		try {
			dcmId = _dcmId;
			nodeId = _nodeId;
			listener = new MyDcmListener(this);
			mySe = new SoftwareElement(listener);
			mySeid = mySe.getSeid();

			cmm = new Cmm1394LocalClient(mySe);
			registry = new RegistryLocalClient(mySe);

			// use cmm to read SDD and to initialize
			// manufacturer, userName, deviceClass
			// now set in the HAVi Registry
			//
			Attribute[] at = new Attribute[10];
			HaviByteArrayOutputStream hbaos = new HaviByteArrayOutputStream();

			hbaos.reset(); hbaos.writeHaviString(manufacturer);
			at[0] = new Attribute(ConstAttributeName.ATT_DEVICE_MANUF, hbaos);

			 hbaos.reset(); hbaos.writeHaviString(userName);
			at[1] = new Attribute(ConstAttributeName.ATT_USER_PREF_NAME, hbaos);

			hbaos.reset(); hbaos.writeInt(deviceClass);
			at[2] = new Attribute(ConstAttributeName.ATT_DEVICE_CLASS, hbaos);

			// … initialization of other Registry attributes

			registry.registerElementSync(0, mySeid, at);

			// now registered so start listening
			//
			mySe.addHaviListener(listener);
		}
		catch(Exception e) {}
	}

	// examples of some DCM APIs
	//
```

```
        public int getDeviceClass() {
            return this.deviceClass;
        }

        public String getUserPreferredName() {
            return this.userName;
        }

        public void setUserPreferredName(String name) {
            this.userName = name;

            HaviByteArrayOutputStream data = new HaviByteArrayOutputStream();
            data.writeHaviString(userName);

            // determine SDD offset
            //
            long offset = 0x0abcd;

            try {
                // write to SDD using the CMM client API
                //
                cmm.writeSync(0, nodeId, offset, data);
            }
            catch(Exception e){}
        }
    }
```

7.6.2 MyDcmListener.java

```
// package com.someone.dcm.test;

import org.havi.types.*;
import org.havi.system.*;

public class MyDcmListener extends HaviListener {
    private MyDcm myDcm;

    // constructor
    //
    public MyDcmListener(MyDcm _myDcm) throws HaviException {
        myDcm = _myDcm;
    }

    // methods
    //

    public final boolean receiveMsg(boolean haveReplied, byte protocolType,
      SEID sourceId, SEID destId, Status state, HaviByteArrayInputStreampayload) {
        if (haveReplied) {
          return false;
        }
      DcmCore core = new DcmCore(protocolType, sourceId, destId, state, payload, myDcm);
      return core.handleRequest();
    }

}
```

7.6.3 DcmCore.java

```
// package com.someone.dcm.test;

import org.havi.types.*;
import org.havi.constants.*;
import org.havi.system.*;
import java.io.*;
```

```
public class DcmCore {
    // fields
    //
    private byte protocolType;
    public SEID sourceId;
    private SEID destId;
    private Status state;
    private HaviByteArrayInputStream payload;
    private MyDcm myDcm;


    // constructor
    //
    public DcmCore(byte _protocolType,
      SEID _sourceId, SEID _destId, Status _state,
      HaviByteArrayInputStream _payload,
      MyDcm _myDcm) {

        protocolType = _protocolType;
        sourceId = _sourceId;
        destId = _destId;
        state = _state;
        payload = _payload;
        myDcm = _myDcm;
    };

    // methods
    //
    public boolean handleRequest{

        try {
            OperationCode opCode = new OperationCode(payload);
            if (opCode.getApiCode() == ConstApiCode.DCM) {

                byte controlFlag = payload.readByte();
                int transferMode = ConstTransferMode.RELIABLE;
                if(controlFlag == 0) {
                    // this is an incoming command
                    //
                    DcmServerHelper dcms = new DcmServerHelper(myDcm.mySe, transferMode);
                    int transactionId = payload.readInt();

                    switch(opCode.getOperationId()) {
                    case ConstDcmOperationId.GET_USER_PREFERRED_NAME:
                        Status returnCode = new Status(ConstApiCode.DCM,
                            ConstGeneralErrorCode.SUCCESS);
                        dcms.getUserPreferredNameResp(sourceId, returnCode, transactionId,
                            myDcm.getUserPreferredName());
                        return true;
                    // case …
                    }
                }
            }
        }
        catch (Exception e){}
    }
  return false;
}
```

7.6.4 DcmCodeUnit.java

```
// import com.someone.dcm.test.*;

import org.havi.system.*;
import org.havi.types.*;
```

```
public class DcmCodeUnit implements DcmCodeUnitInterface {
    private HUID dcmId;

    public int install(GUID nodeId, UninstallationListener listener) {
        // initialize dcmId, the HUID
        MyDcm mdcm = new MyDcm(dcmId, nodeId);
        return 1;
    }

    public void uninstall() {}
}
```

8 HAVi Level 2 User Interface

This chapter provides a specification of the Home Audio/Video Interoperability Architecture User-Interface, (also called the HAVi User-Interface). This HAVi User-Interface is designed as a "TV-friendly" user-interface framework and is explicitly designed to be suitable for use and implementation on a variety of consumer electronic (CE) devices. The application programming interfaces (APIs) for the HAVi User-Interface are contained in the org.havi.ui and org.havi.ui.event packages described in Appendix A: HAVi Java APIs. In case of conflicts between the specification and the Java APIs, the Java APIs shall be the normative reference.

8.1 HAVi User-Interface Design (informative)

The HAVi User-Interface allows applications, written in Java, to determine the user interface capabilities of its host display device, accept input from the user, draw to the screen and play audio clips. It uses a subset of the AWT as defined in the Java 1.1 Core API (reference [8]) and extends this with packages and classes specific to the HAVi platform. This subset is supported in PersonalJava as defined in PersonalJava 1.1 specification (reference [9]).

8.1.1 Remote Control

The user input model from java.awt is extended to support an optional remote control. A large number of events are optional, allowing manufacturers to customize and add value to their products.

8.1.2 Television Specific Support

HAVi also adds classes to support graphics and video display functions that are available in typical television-based systems, including: support for non-square pixels, and graphics / video overlays.

8.2 java.awt Subset

Only a subset of the Java 1.1 java.awt package is required to be present on a HAVi platform. This subset is described within this section in more detail.

8.2.1 Required Elements from AWT

Since most of the widget set in the java.awt package is not "TV friendly", these classes are not required to be present in systems supporting the HAVi UI framework. TV friendly equivalents of these are provided through the HAVi User-Interface framework, which can be extended to support alternative look and feel. Classes of the java.awt package not included in this specification cannot be expected to be present in devices supporting the HAVi User Interface framework. Interoperable HAVi applications shall not make use of these classes. Where an application uses classes which fall outside of the scope of the HAVi specification, the behavior is not determined by this HAVi User Interface specification, rather it shall be determined by the implementation of the underlying platform. This specification does not prevent a manufacturer implementing a particular device using all of AWT, and any applications intended to execute solely in a particular device may exploit any classes or packages known to be in that device, but both the device and application shall not be regarded as interoperable and shall be considered to be proprietary in nature.

The specified set of classes have been chosen such that HAVi applications can implement any missing widget functionality using these classes.

- The main base classes, such as java.awt.Component, are required in order to build the HAVi widgets.
- Other classes, such as java.awt.Color and java.awt.Font, are required for all general drawing and painting.
- The layout classes, such as java.awt.FlowLayout and java.awt.BorderLayout are retained to provide flexible layout of components on various output devices.

The classes from java.awt that are listed in Table 15 are the classes that an HAVi application author can reliably interact with, and use within a HAVi compliant application. Interoperable applications must not use any references from classes in this list to classes not in this list.

Table 15. java.awt Classes Available to Interoperable HAVi Applications

java.awt	java.awt.event	java.awt.image
Adjustable(intf)	ActionListener(intf)	ImageConsumer(intf)
ItemSelectable(intf)	AdjustmentListener(intf)	ImageObserver(intf)
LayoutManager(intf)	ComponentListener(intf)	ImageProducer(intf)
LayoutManager2(intf)	ContainerListener(intf)	ColorModel
MenuContainer (intf)	FocusListener(intf)	DirectColorModel
AWTError	ItemListener(intf)	IndexColorModel
AWTEvent	KeyListener(intf)	MemoryImageSource
AWTEventMulticaster	MouseListener(intf)	PixelGrabber
AWTException	MouseMotionListener(intf)	
BorderLayout	TextListener(intf)	
CardLayout	WindowListener(intf)	
Color	ActionEvent	
Component	AdjustmentEvent	
Container	ComponentAdapter	
Cursor	ContainerEvent	
Dimension	FocusAdapter	
Event	FocusEvent	
EventQueue	InputEvent	
FlowLayout	ItemEvent	
Font	KeyAdapter	
FontMetrics	KeyEvent	
Graphics	MouseAdapter	
GridLayout	MouseEvent	
IllegalComponentStateException	MouseMotionAdapter	
Image	PaintEvent	
Insets	TextEvent	
MediaTracker	WindowAdapter	
Point	WindowEvent	
Polygon	ComponentEvent	
Rectangle	ContainerAdapter	
Shape		
Toolkit		

The classes in Table 15 are not necessarily sufficient to enable a full implementation of a HAVi compliant device, for example a device implementing the HAVi User-Interface could be implemented using Java 1.1, Personal Java , etc., which might require additional requirements on the implementation. The specification is intentionally silent on the mechanisms used to implement the Java environment for a HAVi implementation.

8.2.2 User Input Preference Interfaces

Personal Java 1.1 includes some interfaces which are not found in JDK 1.1 but are useful for a TV friendly user-interface API. The HAVi specification includes a number of interfaces intended to allow Java applications to adapt to mouseless environments like systems operated by remote control. These input preference interfaces allow component developers to specify how users can navigate among and interact with their components. They are only available to components which inherit from org.havi.ui.HComponent. The specification is intentionally silent on the mechanisms used to implement the Java environment for a HAVi implementation. An extra HAVi specific interface org.havi.ui.HAdjustmentInputPreferred is also included.

The org.havi.ui.HNoInputPreferred interface disallows user navigation, and hence actioning, etc.

A component that implements org.havi.ui.HNoInputPreferred indicates that the user may not navigate to this component. However, note that if a component which implements this interface is extended, so that the sub-classed component may implement another "XxxInputPreferred" interface, then in all cases, this other interface may take precedence. In contrast, the method isFocusTraversable shall always return true for components implementing the interfaces org.havi.ui.HActionInputPreferred, org.havi.ui.HAdjustmentInputPreferred, org.havi.ui.HKeyboardInputPreferred, org.havi.ui.HNavigationInputPreferred and org.havi.ui.HSelectionInputPreferred.

The org.havi.ui.HKeyboardInputPreferred interface indicates that it is intended to accept component specific keyboard input from the user. Platforms without keyboards may provide another means for generating such input when this component is edited, for example, by offering an on-screen keyboard.

The org.havi.ui.HActionInputPreferred interface indicates that it is intended to be actioned by the user.

The org.havi.ui.HAdjustmentInputPreferred interface indicates that it is intended to offer increment and decrement functionality to the user.

The org.havi.ui.HNavigationInputPreferred interface indicates that it is intended to offer focus traversal between org.havi.ui.HComponents to the user.

The org.havi.ui.HSelectionInputPreferred interface indicates that it is intended to offer selection and deselection to the user.

8.3 HAVi Extensions to AWT

8.3.1 General API Issues

In this package, passing null to a method or constructor shall generate a java.lang.NullPointerException except in the following circumstances :

- Where null is explicitly documented as being an allowed parameter
- Where the class where the method or constructor is defined inherits from java.util.EventObject or java.lang.Exception

It is an allowable option to override protected, public and package level methods in HAVi using the standard java inheritance conventions.

8.3.2 User Input

Java Applications in HAVi can accept input from a keyboard, a mouse or a remote control. The

keyboard and mouse inputs are supported by functions in the java.awt and java.awt.event packages. Remote control input is provided with classes in the org.havi.ui.event package. The org.havi.ui and org.havi.ui.event packages include classes that allow the application to determine the user-input capabilities of the platform on which the application is running.

8.3.2.1 Remote Control Support

The HAVi remote control classes are extended from the java.awt.event key event classes. All of the events that are added for the remote control are optional. The remote control keys fall into two categories: colored keys and dedicated keys. The intention of these keys is to provide the user direct access to various functions; however, the platform *may* implement a virtual (on-screen) mechanism to generate these events, but shall take care in this case not to hide the application. Note that it is an implementation option if (remote control) key events are repeated.

8.3.2.1.1 Remote Control Colored Keys

Up to six colored soft keys can be included on a remote control. These are optional, and are to be identified with a color. If implemented, these keys are to be oriented from left to right, or from top to bottom in ascending order. The application can determine how many colored keys are implemented, and what colors are to be used, so that the application can match the controls.

The following identifiers are available for colored key events: VK_COLORED_KEY_0, VK_COLORED_KEY_1, VK_COLORED_KEY_2, VK_COLORED_KEY_3, VK_COLORED_KEY_4, VK_COLORED_KEY_5.

8.3.2.1.2 Remote Control Dedicated Keys

The org.havi.ui.HRcEvent class defines a number of dedicated remote control events that can be used by applications. Although none of the events in the org.havi.ui.event.HRcEvent class are required to be implemented, events for power (VK_POWER), volume up and down (VK_VOLUME_UP and VK_VOLUME_DOWN), and channel up and down (VK_CHANNEL_UP and VK_CHANNEL_DOWN) are highly recommended.

However, whilst the dedicated remote control events are themselves device independent, the precise set of dedicated keys that is implemented is device dependent. The org.havi.ui.event.HRcCapabilities class enables an application to discover which events are implemented and how these are to be labeled to match the platform implementation.

8.3.2.2 Keyboard

HAVi supports keyboards via the java.awt.event package. The events supported on a keyboard can be determined by the org.havi.ui.event.HKeyCapabilities class.

Note that systems that do not include a physical keyboard can check each component to see if it implements org.havi.ui.HKeyboardInputPreferred. If this interface is implemented, the system may enable user input of alphanumeric key events, for example, via a “soft” on-screen keyboard.

8.3.2.3 Mouse

Mouse support is optional. The presence of a mouse can be detected with the org.havi.ui.event.HMouseCapabilities class.

Mouse functionality is provided by the java.awt.event package. HAVi applications must be written in such a way that a free roaming cursor is not required for correct operation. This does not mean that a HAVi application could not implement, e.g. a drawing program, but rather that the user should not be able to put the application into a state that cannot be exited without a mouse. (A user-friendly drawing package would also notify the user that a mouse is required to use this application properly.)

8.3.2.4 User Input Capabilities

Three classes are available to determine the capabilities of the user input for a given platform: org.havi.ui.event.HKeyCapabilities, org.havi.ui.event.HMouseCapabilities, org.havi.ui.event.HRcCapabilities. Each of these classes includes a method called getInputDeviceSupported, which returns true if the particular device is known to be available.

8.3.2.5 User Input Representation

The org.havi.ui.event.HRcCapabilities class includes a method called getRepresentation, which returns an object of type org.havi.ui.event.HEventRepresentation. This class defines an event as having a known representation as a string, color or symbol, or having no supported representation. The particular text, color, or symbol can be determined by calling getString, getColor or getSymbol respectively. This allows an application to describe a button on an input device correctly for a given platform. All available events should have a text representation from getString.

The six colored key events (VK_COLORED_KEY_0 -- VK_COLORED_KEY_5), if implemented, must also be represented by a color – the getColor method returns a java.awt.Color object.

Key events may also be represented as a symbol – if the platform does not support a symbolic representation for a given event, then the application is responsible for rendering the symbol itself. Application rendering of keys without a symbolic representation, but with a commonly known representation, should follow the guidelines as defined in the Javadoc definition of the class.

8.3.3 Graphics Devices and Configurations

8.3.3.1 Background

There are some specialized requirements for running applications within a consumer electronic environment, rather than the simpler situation that occurs when an Applet is displayed within a web-browser. Most notably the screen dimensions and aspect ratios are significantly different between PCs and CE devices. In the current on-screen display (OSD) graphics model of today's set-top box units, video may be output in a number of different configurations, e.g. traditional 4:3 TV display, or 16:9 widescreen TV displays, etc. The graphics resolution and aspect ratio are often locked to the video resolution and aspect ratio. If the video aspect pixel ratio changes then the graphics pixel aspect ratio may also change. Thus, there are requirements to:

- Determine the resolution and physical characteristics of the current display device.
- Detect modifications to the resolution and physical characteristics of the current display device.

8.3.3.2 The HAVi Screen Reference Model

HAVi provides a model for the video output from a consumer electronics device. Instances of the class HScreen represent each independent final video output signal from a device. Each independent final video output signal is made up from the sum of graphics devices, video devices and backgrounds. These are represented by instances of the classes HGraphicsDevice, HVideoDevice and HBackgroundDevice respectively. All of these classes inherit from a common parent class - HScreenDevice.

The HAVi User-Interface specification provides limited support for applications to be displayed so that they are split across multiple concurrent display devices – the HSceneFactory class allows the HGraphicsDevice to be specified in the HSceneTemplate used to generate the HScene's for the application.

8.3.3.3 The HAVi Screen Device Discovery Classes

HAVi defines a means to allow applications to discover the range of display devices available. The model followed by HAVi is based on the model used in Java2 as described by the following three classes in the java.awt package - GraphicsDevice, GraphicsConfiguration and GraphicsConfigTemplate. In HAVi, this model is generalized to apply to video devices and to background devices.

8.3.3.3.1 Querying the Configuration of a Display Device

For each display device class (HVideoDevice, HGraphicsDevice and HBackgroundDevice), there are classes whose name ends in "Configuration" which represent distinct possible configurations of a single device. Applications may obtain a list of all possible configurations of a particular device. Applications may also obtain the current configuration using the getCurrentConfiguration method. Subject to security and resource management issues, applications may also set the configuration of a device using methods found on each device class.

Applications that are interested in a particular configuration of a device can request configurations matching a specific set of constraints. The first step in this process is to construct objects whose name ends in "ConfigTemplate". Instances of these classes can then be populated with the properties by the application and then used to request a configuration supporting those properties. Properties can also have priorities attached to allow applications to express whether support for that property is required by the application, whether support for that property is only preferred by that application, whether support for that property is required to be absent or whether support for that property is preferred to be absent. In some cases, properties such as PIXEL_ASPECT_RATIO require extra information. This extra information can be provided as part of the method used to add the property to the configuration template.

The Configuration for a Device can be acquired, using the getCurrentConfiguration method. A description of this Configuration can be obtained using the getConfigTemplate method that yields a ConfigTemplate that uniquely identifies the given Configuration. Individual properties in this ConfigTemplate can then be examined using the getPreferencePriority and getPreferenceObject methods – features that are implemented will return REQUIRED, features that are not implemented will return REQUIRED_NOT. Values of some properties may also be obtained through a limited set of query methods provided on HScreenConfiguration.

8.3.3.3.2 Compatibility with Existing java.awt Methods

The java.awt.Toolkit.getScreenSize method shall be equivalent to the pixel resolution of the current configuration of the default screen device returned by HScreen.getDefaultGraphicsDevice.

The value of the java.awt.Toolkit.getScreenResolution method is implementation specific. This method shall not be used by applications.

Where the screen aspect ratio is unknown (such as in the case where a set-top box is connected to an analog display), the default aspect ratio is 4:3. In the case where an analog monitor is used with a HAVi compliant set-top box the resolution returned shall be based on the raster of the set-top box, ignoring any interpolation or other processing that may be present in the monitor.

The java.awt.Toolkit.getNativeContainer method shall return null; interoperable applications should not rely on this method.

Where an input parameter to a method call is specified to be more restrictive than its Java type allows (e.g. only a restricted set of numbers are allowed as inputs), providing values outside the allowed range shall result in a java.lang.IllegalArgumentException being thrown.

8.3.3.4 Detecting Configuration Changes on a Display Device

It is important for CE devices to be able to detect variations in their settings, since they may be subject to "on-the-fly" modifications of these settings, for example, they may be heavily influenced by the nature of some input video streams. Hence, the HScreenDevice class provides support for detecting when its configuration (settings) have been changed, using the HScreenDevice.addScreenConfigurationListener methods and the HScreenConfigurationListener and HScreenConfigurationEvent classes.

When an HScreenDevice's configuration is modified, then an HScreenConfigurationEvent is generated. Note that after a HScreenConfigurationEvent is obtained any HScreenConfiguration (or HScreenConfigTemplate) associated with that HScreenDevice must be reacquired to obtain the current settings for the device.

In general, a modification to the HScreenDevice might require that the displayed user-interface be modified, e.g. if the resolution has changed, or the pixel aspect ratio has been modified.

8.3.3.5 Emulated Display Devices

The HAVi User-Interface introduces extra sub-classes that are used to indicate that a device may perform emulations of other device capabilities:

- HEmulatedGraphicsDevice
- HEmulatedGraphicsConfiguration

Instances of these classes can be returned by the same methods that would return the corresponding class without "Emulated" in the class name. Returning the sub-class indicates that the implementation is emulating the requested configuration on one of its actual supported configurations. The class HEmulatedGraphicsConfiguration includes methods to allow applications to compare the configuration being emulated and the actual underlying configuration being really used. The extent of support for emulated configurations is a profile issue. All possible emulated configurations are not required, or guaranteed to be supported. Emulated configurations may have a significant performance penalty with respect to those supported natively on the device.

8.3.3.5.1 Mapping from Authoring to Device Coordinates

A special case of device emulation is the emulation of various graphics coordinate systems on a

single physical device. The HAVi User-Interface provides mechanisms that allow devices to perform such emulations, e.g. by down-sampling a high-resolution system to match the limitations of a standard definition display. Thus, authors can rely on seamless mapping between authoring and device coordinates by the use of the HEmulatedGraphicsDevice class. Authors may determine an appropriate graphics device and request the best configuration that matches their requirements, as in a standard device discovery mechanism – or examine configurations themselves (both emulation and implementation) to determine appropriate settings. The extent to which devices are required to support emulation of other coordinate systems is profile dependent.

8.3.3.6 Integrating HAVi Video Support into Platforms

The HAVi specification includes several classes to represent video in the user interface system. This representation of video devices only includes the display of video. The setup of the video decoder and the video pipeline is not included in this specification.

The class HVideoComponent is intended to be returned by a platform specific controller for video. In platforms based on the Java Media Framework, the Player.getVisualComponent method shall return objects of this class. The class HVideoDevice provides two hooks to platform specific APIs for this setup, the methods getVideoController and getVideoSource. On platforms based on the Java Media Framework (JMF), the getVideoController method shall return a JMF Player. The getVideoSource method shall return a platform specific class encapsulating a reference to the source of the video. Possible examples of the class to be returned here could include java.net.URL or javax.media.MediaLocator.

8.3.3.7 Backgrounds

The HAVi specification includes several classes to represent the background of a screen, i.e. the area that is behind the running graphics and video and not covered by those. Using the same naming convention as video and graphics, these are called HBackgroundDevice, HBackgroundConfiguration and HBackgroundConfigTemplate. The basic HBackgroundConfiguration allows applications to control a single full screen background color.

The HAVi specification includes support for more sophisticated backgrounds - still images. Applications wishing to use these shall request an HBackgroundConfiguration supporting them by using the STILL_IMAGE property in an HBackgroundConfigTemplate. If this feature is supported by the platform concerned, when such a configuration is requested, an instance of the class HStillImageBackgroundConfiguration shall be returned. This class adds two extra methods, over the standard background configuration, which support the loading of background images. This loading is done through the HBackgroundImage class.

Using the HBackgroundImage class rather than the standard java.awt.Image allows for image formats that are decoded using hardware outside of the graphics system. One specific example of this is the decoding of still MPEG I frames using an MPEG video decoder. This is a commonly used feature in some devices since it provides good quality backgrounds without using software decoders or system memory. In systems where the same underlying MPEG video decoder can be used to decode both video and MPEG I frames, this decoder shall be represented both by an HVideoDevice instance and by an HBackgroundDevice instance for each application. Where an HBackgroundDevice and an HVideoDevice both map onto the same underlying real resource in the device, the ZERO_VIDEO_IMPACT property in HBackgroundConfigTemplate shall be used to discover and limit any impact of one on the other.

- An application requesting a HBackgroundConfiguration using a HBackgroundConfigTemplate containing the ZERO_VIDEO_IMPACT property with priority REQUIRED shall only be returned one which would have absolutely no impact on any HVideoDevice if that HBackgroundConfiguration was set for its HBackgroundDevice. For example, only systems where a separate decoder is used for HBackgroundImages from video shall return a HBackgroundImage for such a template.

- An application requesting a HBackgroundConfiguration using a HBackgroundConfigTemplate containing the ZERO_VIDEO_IMPACT property with priority PREFERRED shall only be returned one which would have no permanent impact on any HVideoDevice if that HBackgroundConfiguration was set for its HBackgroundDevice. For example, systems where the same underlying hardware is used for decoding both video and HBackgroundImages but where once decoded, the HBackgroundImage is copied into a separate decoder memory from video and video decoding resumes, shall return an HBackgroundConfiguration in this case but not the previous case.

- Implementations where decoding of HBackgroundImages interrupts the video for the duration of the still image shall not support any HBackgroundConfiguration where ZERO_VIDEO_IMPACT is either REQUIRED or PREFERRED.

On implementations where the same underlying MPEG video decoder is used for both video and HBackgrounds, the most recent request of an application shall always be granted where the single underlying decoder is already being used by that application.

8.3.3.8 Control of Screen Configurations

The HAVi specification is silent about whether a single display device is shared between multiple applications or not. For the case where a display device may be shared between applications, it provides a mechanism for applications to assert control over the right to change the configuration of the display device. The HScreenDevice class includes methods to allow applications to reserve and release the right to control this configuration. It also allows an application to register and remove listeners for events that are generated when the applications reserve and release this right.

Applications wishing to be able to control the configuration of an HScreenDevice must define a class implementing the ResourceClient interface and pass an instance of this class to the reserveDevice method of the HScreenDevice that they wish to control. If the reserveDevice method succeeds then the application obtains control over the device configuration. When an application calls the HScreenDevice.getClient method this will return the ResourceClient passed in to the last call to the reserve method on that HScreenDevice instance.

Where there is a conflict between applications, this specification includes a mechanism to allow the platform to arbitrate between conflicting applications. The policy for this arbitration is intentionally not defined in this specification. When it is decided to remove the right to control a screen from an application, this is notified through the ResourceClient interface, the notifyRelease method will always be called. The requestRelease method will only be called when the existing owner of the resource and the application requesting the resource are authenticated to have a secure relationship of some form. This specification is silent about the details of this authentication.

8.3.4 Graphics and Video Integration

8.3.4.1 Configurations

The HAVi specification allows applications to express the relationship between video, graphics and backgrounds. The method HScreen.getCoherentScreenConfigurations allows applications to express a common set of constraints for video, graphics and backgrounds and get back a coherent answer.

In addition to this, there are several means to express constraints between video and graphics. These can be used for applications which already have running video to fit a graphics configuration to that video or which have already running graphics to fit video to that graphics. In HGraphicsConfigTemplate, the constant VIDEO_MIXING allows applications to request configurations where graphics is super-imposed above video but without any requirement for pixels to be aligned. In HScreenConfigTemplate, there are constants to allow applications to ask for configurations as follows:

- VIDEO_GRAPHICS_PIXEL_ALIGNED - video & graphics pixels are the same size and aligned
- ZERO_VIDEO_IMPACT - a new graphics configuration must not change the existing video configuration
- ZERO_GRAPHICS_IMPACT - a new video configuration must not change the existing graphics configuration

8.3.4.2 Coordinate Spaces

The HAVi specification includes a normalized screen coordinate system that represents the coordinates on the screen as floating point numbers between zero and one. This coordinate system is not pixel based. Such a non-pixel-based coordinate system enables the following:

- meaningful results, even when the graphics configuration has not been determined
- meaningful results when presented video does not have a java.awt component.
- meaningful results when the video display and the graphics display are not necessarily aligned / share the same origin / share the same resolution, etc.

This screen-based coordinate system is encapsulated in the HScreenRectangle and HScreenPoint classes. This specification is silent about conversion between normalized and video coordinates. This should be addressed as part of the API providing support for control of video.

For graphics, these conversion mechanisms are found on the HGraphicsConfiguration class, since the conversion from screen to graphics coordinates is dependent on the current graphics device settings (Configuration) – especially if e.g. the graphics resolution can be varied independently of the video resolution, etc.

The HScreenRectangle mechanisms can be used to enable the alignment of (portions of) video and (portions of) graphics. The HScreenLocationModifiedListener and HScreenLocationModifiedEvent allow mechanisms to determine if the on-screen location of an HVideoComponent is modified (rather than its relative location within its enclosing container).

8.3.4.3 Transparency between Graphics and Video

The HAVi specification includes support for applications to request transparency between graphics and video. This is provided by the getPunchThroughToBackgroundColor method on the HGraphicsConfiguration class. These methods provide a factory that enables applications to provide an opaque java.awt.Color and obtain a java.awt.Color supporting some form of transparency between graphics and video. These Color objects may be used in the drawing methods in the java.awt.Graphics class to cause video to appear in the graphics system.

8.3.5 HSceneFactory, HSceneTemplate and HScene

The HAVi User-Interface is deliberately agnostic concerning the implementation of a "coordinating" environment that provides the mechanism for a user to choose and run one or more applications. In HAVi, this "coordinating" environment is known as a "home navigation shell". Application writers cannot make any assumptions that their application GUI will always be immediately visible. For example, valid implementations of a coordinating environment might include:

- A simple "full-screen" view on a single application at any one time (with some undefined mechanism to switch between them).
- A multi-window system, where windows may obscure each other.
- A "paned" system where each application occupies an area on-screen – i.e. each application is always visible, but they may be resized if other applications are installed.

Thus, mechanisms are required to initiate an on-screen display and to indicate "user-interest", or other modifications to the rendering area. These mechanisms should:

- Enable an application to request an area on-screen – however, given the possibility of differing styles of coordinating environment, an application cannot reasonably expect that its request will always be honored perfectly, and thus, a mechanism is required to indicate preferences for the application location "on-screen".
- Indicate whether the current application is the one which the user is specifically interested in. For example, a "well-behaved" application which the user is not currently using might release, or reduce its consumption of any limited resources.
- Indicate to an application that its extent and position on-screen have been modified somehow by the home navigation shell.
- Allow an application to indicate to the system, that it requires the user's attention. For example, the system may either indicate to the user that the user should choose the indicating application, or might simply automatically switch to that application.

8.3.5.1 Requesting an Area On-screen

8.3.5.1.1 HSceneFactory and HSceneTemplate

The HSceneFactory is a factory class that is used to generate HScene objects. An application can indicate the location and dimensions of the HScene in the associated HSceneTemplate, although it is not guaranteed that the resulting HScene will necessarily match all of these preferences – since this

is dependent on the implementation of the controlling shell and its associated policies, etc.

The application should call HSceneFactory.resizeScene if it wishes to re-size the HScene.

8.3.5.1.2 HScene

An HScene is an HContainer representing the displayable area on-screen within which the application can display itself and thus interact with the user. However, HScene does not paint itself on-screen, only its added "child" components and hence there is no requirement to allocate "pixels" to the HScene directly – its only effect is to "clip" its child components. Hence, HScene may be regarded as a simple connection to the window management policy within the device, acting as a "screen resource reservation mechanism" denoting the area within which an application may wish to present a component, at some point in the future. Since an HScene is by definition not painted, i.e. it is effectively transparent, the area behind (all) HScene's in the z-ordering may be exposed by the platform as an HBackgroundDevice, and/or HVideoDevice's. However, HAVi does not require platforms to provide such device capabilities, this is platform specific. The HScene semantics for transparency need to be specified exactly on a per-platform basis, for example, on some platforms an HScene might be transparent to other HScene's due to other separate applications.

For all interoperable applications, the HScene is considered the main top-level component of the application. No parent component to an HScene should be accessible to applications. Interoperable applications should not use the getParent method in HScene, since results are implementation dependent and valid implementations may generate a run-time error.

In terms of delegation, the HScene shall behave like a java.awt.Window with a native peer implementation, in that it will not appear to delegate any functionality to any parent object. Components which do not specify default characteristics inherit default values transitively from their parent objects. Therefore, the implementation of HScene must have valid defaults defined for all characteristics, e.g. Font, foreground Color, background Color, ColorModel, Cursor and Locale.

The HScene has a null LayoutManager by default – all widgets are placed using an X, Y co-ordinate, specified by the widget.

When created an HScene is not initially visible, and a call to setVisible is required to display the HScene (and also to hide it).

The application should call HSceneFactory.dispose if it wishes to destroy the HScene (and all of its currently added Components) and therefore release their associated resources for future garbage collection by the platform.

8.3.5.2 Modifications to the HScene: Focus and Resize events

The HScene object accepts java.awt.event.WindowEvent's, and interprets them as a java.awt.Window, however it is not required for the home navigation shell to generate all types of java.awt.event.WindowEvent.

Applications can use the java.awt.Component.requestFocus method on the HScene to indicate to the home navigation shell that the HScene should be receiving input focus. This request should be treated as a request to make the entire application visible and ready for user input, e.g. by expanding an icon, or changing the stacking order between competing overlapping applications. The decision as to whether or whenever the HScene (application) gains the input focus is entirely platform specific in terms of policy, etc. The java.awt.Component must be visible on the screen for this request to be granted – note that visibility in this context refers to whether the application has called the HScene.setVisible method, rather than any possible non-application-defined-behavior, due

to the action of the coordinating shell hiding an application.

The java.awt.event.ComponentEvent's COMPONENT_MOVED and COMPONENT_RESIZED will be received by the HScene when the controlling shell has modified the position of the HScene or changed its dimensions on screen, respectively.

8.3.5.3 Application "user-interface" Lifecycle

- Outside the scope of the HAVi User-Interface:
  - The application is acquired by the platform.
  - The application is validated and security checked (possibly including authentication, byte-code verification, etc.).
  - The virtual machine is initialized, ClassLoader created, etc.
  - The application is executed
  - If the application does not require a user-interface, then it may continue as per normal.
- If a user-interface, and hence some screen resource is required, then the application traverses the HScreen, HGraphicsDevice, HGraphicsConfiguration space to determine an appropriate configuration, using HGraphicsConfigTemplate e.g. video-mixable, full-screen graphics, square pixel aspect ratio, resolution 1280 by 1024.
- The application configures the HGraphicsDevice appropriately, using the setGraphicsConfiguration method.
- The application requests that the HSceneFactory effectively grant it access to part of the screen for that device, using HSceneTemplate, e.g. full-screen display.
- The HSceneFactory returns an appropriate HScene container within which the application can display itself.
- The application uses the HScene container to add all of its components to make its user-interface.
- The application may take advantage of java.awt.WindowEvent's, to determine whether it has the user's (input) focus.
- The application may take advantage of the events COMPONENT_RESIZED and COMPONENT_MOVED, to determine when its HScene extent / location has been modified and to tailor its presentation accordingly.
- The application resizes the HScene by using HSceneFactory.resizeScene – if it wishes to resize the HScene, with the caveat that this may not be allowed by the external environment, e.g. due to window-manager policy, etc.
- The application terminates its on-screen presentation by calling the HScene.dispose method
- Outside of the scope of the HAVi User-Interface:
  - The application itself terminates.

8.3.6 Effects and Visual Composition using Component Mattes

8.3.6.1 Component Mattes

With org.havi.ui, the user interface is constructed from a set of components arranged in a hierarchy. The root of the hierarchy is an instance of HScene, leaf nodes are instances of HComponent and intermediate nodes are instances of HContainer. Components within a container are ordered from back to front. An example is shown in Figure 39, where *c3*, a container, is the back most component and *c1* the front most.

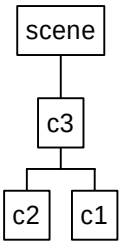

Figure 39. Scene Hierarchy

With the HMatte interface, the scene hierarchy can be modified by the inclusion of mattes (additional alpha sources), potentially for each member, i.e.:

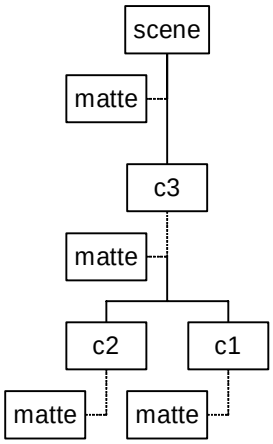

Figure 40. Scene Hierarchy with Mattes

The mattes influence the rendering of the scene, their operation can be visualized using a 2½D or layering model. The example below corresponds to the hierarchy in Figure 40 (for simplicity, the scene matte is not shown).

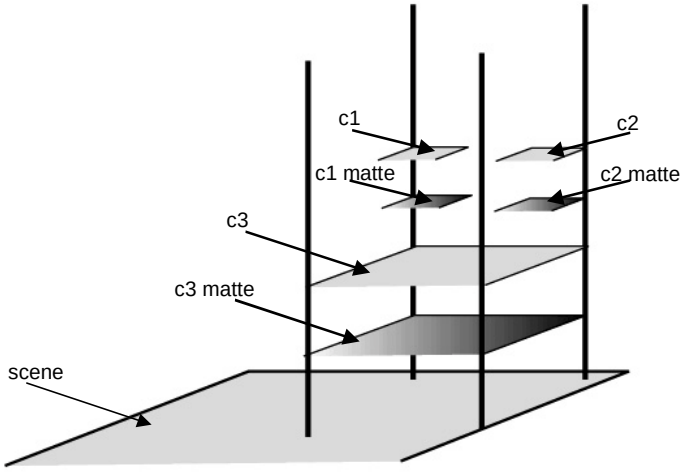

Figure 41. Component Mattes

Where pixels in a component already have an alpha value (e.g., from a PNG image), the alpha value from the component and the alpha value from the matte are multiplied together to obtain the actual alpha value to be used for that pixel.

8.3.6.2 Component Grouping

A container is either “grouped” or “ungrouped”. When a container is ungrouped, its matte only influences the appearance of those regions of the container not covered by members of the container (i.e., exposed regions of the container’s background). When a container is grouped, its matte influences the appearance of its background and all members of the container. For example, grouping a container and setting its matte to indicate 50% transparency will fade the container’s background and all members of the container. If it is ungrouped only the background will fade.

An HContainer may be rendered as follows:

- If the container is ungrouped, the container’s background is first rendered and then composited with the container’s matte (i.e., the RGBA value of the container’s background is combined with the alpha value from the matte). Then, in back to front order, each member of the container is rendered, composited with its matte, and then composited with the container.

- If the container is grouped, the container’s background is first rendered. Then, in back to front order, each member of the container is rendered, composited with its matte, and then composited with the container. The result is then composited with the container’s matte.

After an HContainer is rendered, it is composited with its parent. Compositing of an HScene is determined by the configuration of display devices.

8.3.6.3 Examples of Mattes and Component Composition

a) Bar matte for lower component.

b) As in a), with top components grouped to lower.

c) Circular mattes for top components.

d) As in c), with bar matte for lower component.

e) As in d), with top components grouped to lower.

Figure 42. Visual Composition Examples

8.3.6.4 Effects

A great variety of effects (e.g., wipes and fades) can be performed by using *matte animations* –

sequences of mattes where the "active" element is changed over time. Matte animations can be combined with other techniques, such as component movement, to produce additional effects. The construction of matte animations is facilitated by the following classification of mattes:

HFlatMatte – the matte is constant over space and time, it can be specified by a float (0.0 is fully transparent and 1.0 fully opaque)

HImageMatte – the matte varies over space but is constant over time, it can be specified by an "image mask" (a single channel image) where the pixels indicate matte transparency

HFlatEffectMatte – the matte is constant over space but varies over time, it can be specified by a sequence of floats

HImageEffectMatte – the matte varies over space and time, it can be specified by a sequence of image masks

8.3.6.5 Matte Sizes and Offsets

When a HImageMatte or HImageEffectMatte is assigned to a component, the associated image (or images) is by default aligned with the component so that their origins – the pixel at (0,0) – coincide. The offset of the matte with respect to the component can be altered using the setOffset method of HImageMatte and HImageEffectMatte. Regions of the component outside the matte (resulting from either a matte being smaller than the component, or from shifting the matte) are not matted.

8.4 HAVi Widget Framework

The HAVi widget framework is designed to allow maximum flexibility to implementers of applications. It also provides the necessary extensibility to allow the widget framework to be used as the basis for other application types, such as broadcast applications. By default, the HAVi widget framework only copies object references, and does not clone objects. Cases where objects are clone'd shall be marked explicitly.

8.4.1 HAVi Event Mechanism

The HAVi event mechanism is composed of the seven classes listed in Table 16:

Table 16. HUI Events

Event	Use
HActionEvent	Interact with a component implementing the HActionInputPreferred interface
HFocusEvent	Interact with a component implementing the HNavigationInputPreferred interface
HRcEvent	Provide remote control event capabilty
HKeyEvent	Interact with a component implementing the HKeyboardInputPreferred interface
HAdjustmentEvent	Interact with a component implementing the HAdjustmentValue interface
HItemEvent	Interact with a component implementing the HSelectionInputPreferred interface
HTextEvent	Interact with a component implementing the HKeyboardInputPreferred interface

These classes serve as the mechanism by which HAVi components inform each other of event occurrences. They are not intended to be generated from applications.

A HAVi widget must respond to these events in addition to other applicable user-input mechanisms. However, interoperable widgets must not respond to specific key codes received through the Java AWT KeyEvent mechanism.

HXXXEvents are generated and dispatched by the HComponent base class. For example, this class must intercept suitable Java key events and generate HKeyEvents from them. This means that widgets will receive two events - the original KeyEvent and a new HKeyEvent. Although it is possible to "discover" the platform-specific implementation of HXXXEvents via this mechanism, interoperable widgets may not use this information. Widgets may ignore the KeyEvent in favor of only handling the HKeyEvent.

8.4.2 Abstraction of “Feel”

In order to provide the necessary flexibility, the HAVi User-Interface widget framework is defined around a core of abstract *Component Behaviors*. These effectively define the functionality (or “feel”) of each widget which is derived from one of the Component Behaviors. Behaviors are defined for widget types having a number of states, which may be used to mimic the behavior of typical widgets.

In summary these Component Behaviors are:

- HVisible – Behavior providing basic display functionality.
- HNavigable– Behavior enabling widgets to receive navigational focus, and to define some kind of display change associated with focus change.
- HActionable– Behavior providing functionality to be invoked in response to an action.
- HSwitchable– Behavior allowing a widget to be actioned and to retain internal state information in addition to simple action behavior.
- HAdjustmentValue, HItemValue, HTextValue - Behaviors permitting the definition of widgets that return values to applications in response to user interaction.

Based upon these fundamental abstract Behaviors, all necessary HAVi functionality can be provided through derived concrete widgets, either for the provision of HAVi specific user-interfaces, or for HAVi specific widgets. In addition, these abstractions form the basis upon which other interactive applications may be built without the requirement for the use of HAVi specific widgets. Thus, the HAVi widget framework is more generally applicable to interactive application execution, rather than exclusively focused upon HAVi. The Javadoc describes these states in more detail, including any valid DISABLED states.

8.4.3 Framework Class Hierarchy

The HAVi widget framework consists of a base class (HVisible) and a set of interfaces that model the behaviors different types of widget may exhibit. The behavior is modeled on the number of states a widget may represent. For each such state a widget can present a particular representation (graphical, textual and sound) to the user.

The widget framework allows for simple user interface development by application authors. It also reduces the size of the developed application, since most of the presentation and interaction

capability is resident on the device – developers can concentrate on the specific functionality of their application.

8.4.3.1 HContainer

Components in the HAVi User-Interface are explicitly allowed to overlap each other. Hence, the HAVi User-Interface extensions adds additional Z-ordering related methods to org.havi.ui.HContainer:

Additional semantics related to transparency of the HContainer itself and its Components, are also defined via the HMatteLayer interface.

The org.havi.ui.HContainer class also adds the ability to determine whether hardware double buffering is present, using the isDoubleBuffered method.

The org.havi.ui.HContainer class also adds the ability to determine whether it is completely opaque, by applications overriding the isOpaque method.

Additionally, the default LayoutManager for HContainer is defined to be null, i.e. absolute positioning, in contrast to the FlowLayout used in java.awt.Container.

8.4.3.2 HComponent

The base class for all HAVi widgets.

The org.havi.ui.HComponent class extends java.awt.Component to include additional semantics related to transparency of the HComponent, defined via the HMatteLayer interface.

The org.havi.ui.HComponent class also adds the ability to determine whether hardware double buffering is present, using the isDoubleBuffered method.

The org.havi.ui.HComponent class also adds the ability to determine whether it is completely opaque, by applications overriding the isOpaque method.

8.4.3.3 HVisible

Represents a widget that has only two states, for example HStaticText or HStaticIcon. This widget can be in either a “normal” state or a “disabled” state.

8.4.3.4 HNavigable

An interface that is implemented by classes that are derived from HVisible for adding an additional state that is used to indicate if the widget is currently focused.

The HNavigable interface also provides the functionality necessary to manage the focus navigation between widgets assuming a remote control style UP, DOWN, LEFT, RIGHT form of navigation, using the setFocusTraversal method.

The precise semantics of the HNavigable interface are defined in the supporting Javadoc.

8.4.3.5 HActionable

The HActionable interface extends HNavigable by adding an additional state that is used to indicate when the widget has been actioned.

The HActionable interface provides the functionality necessary to associate HActionListeners with the widget, using the addHActionListener and removeHActionListener methods. These HActionListeners will be called when the widget is actioned.

A widget that implements the HActionable interface is actioned when it receives a havi.ui.event.HActionEvent key event. The widget will move into its Actioned state by presenting its Actioned look. Any associated HActionListeners will be called by the widget calling its HActionInputPreferred.processHActionEvent method. When the HActionListeners have returned the widget will return to its focused state.

The precise semantics of the HActionable interface are defined in the supporting Javadoc.

8.4.3.6 HSwitchable

The HSwitchable interface extends HActionable by adding an additional state that is used to maintain an internal (on/off) value.

The state transitions for HSwitchable are as follows:

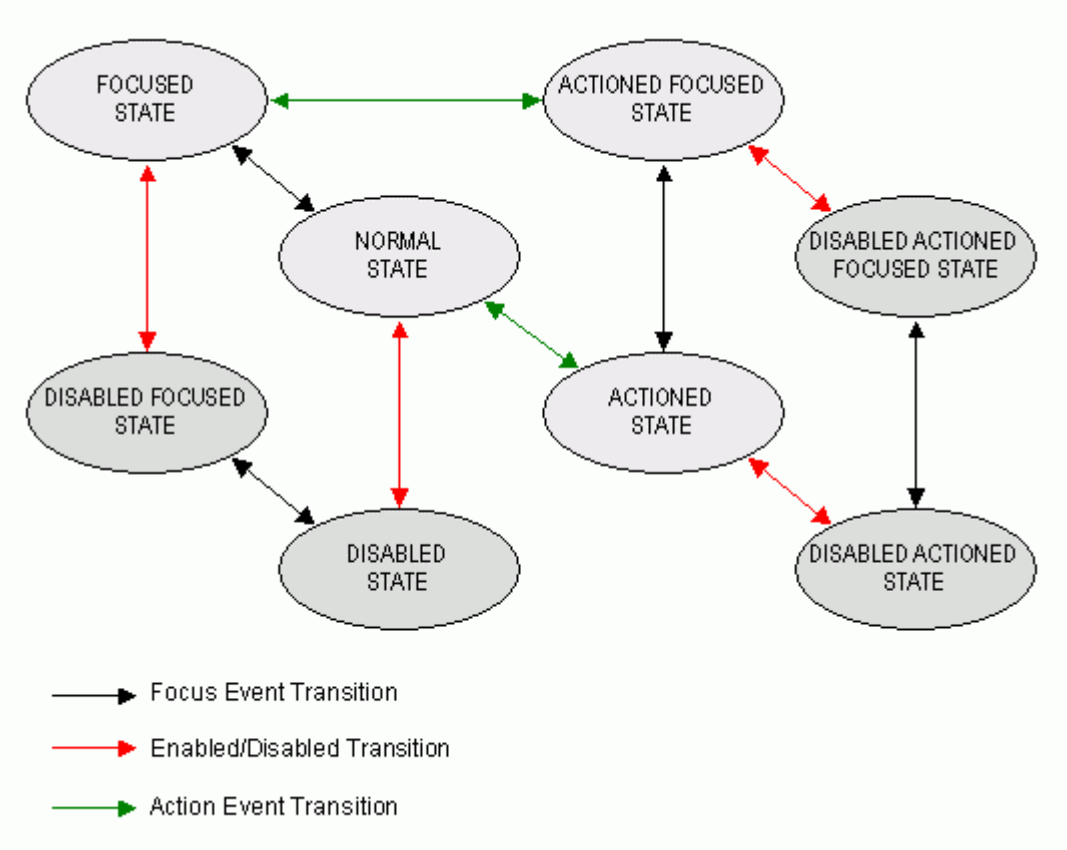

Figure 43. HSwitchable Transitions

The precise semantics of the HSwitchable interface are defined in the supporting Javadoc. Note that any state is permitted to change to the “disabled” state.

8.4.3.7 HAdjustmentValue, HItemValue, HTextValue

These interfaces extend HNavigable by adding support for managing a widget with an internal value that can be manipulated by user interaction.

All HAVi UI components that require adjustable numerical values, such as range controls, have implemented the HAdjustmentValue interface. Here, the value responds to unit & block increments, and has an optional sound associated with such adjustments.

All HAVi UI components that have selectable content, such as list groups, have implemented the HItemValue interface. An optional sound can be associated with item selection.

All HAVi UI components that have editable text content, such as text entry controls, have implemented the HTextValue interface.

The precise semantics of the HValue interface are defined in the supporting Javadoc.

8.4.4 Separation of "Look"

The flexibility of the HAVi widget framework is further enhanced by separating the "look" component from that for "feel". This allows easy construction of many styles of presentation associated with each of the abstract Component Behaviors defined previously.

Content can be associated with each state of a widget. For each widget state, textual, graphical and user defined content can be associated with the widget. The HLook interface defines the mechanism by which the content for the particular state of the widget can be rendered.

The HLook method showLook is used to provide the rendering of the content for the widget. Note that since this method is separated from the widget class, there is no need to subclass the widget to change its look. The showLook method is responsible for repainting the entire component, including its background, subject to the clipRect of the Graphics object passed to it. The showLook method should not modify the clipRect of the Graphics object that is passed to it.

An HLook may also provide some form of border decoration, for example, drawing a rectangle around the widget when it has focus. To allow for predictable layout and presentation the HLook interface provides methods that are used to indicate the size of such a border area.

To support layout managers the HLook interface also defines the following methods, which allow the associated HVisible to query the HLook for its maximum, minimum and preferred sizes: getMaximumSize, getMinimumSize, getPreferredSize.

8.4.5 Pluggable Looks

The HAVi Widget framework provides a set of standard classes that implement the HLook interface. These can be regarded as the set of default looks that will be provided by all implementations. The particular rendering of a look is not defined and is manufacturer dependent.

Pluggable Look is defined in such a way as to allow implementers to extend the number of "looks" available to their application. By doing so, new "looks" are automatically available for every Component Behavior and for all widget types derived from those Behaviors. This is described in the Pluggable Look Interface.

To facilitate application development and to limit the size of applications, a set of pre-defined "looks" is provided. These are:

- HAnimateLook – presentation of an animated image sequence.
- HGraphicLook – presentation of graphical content.
- HRangeLook – presentation of a value within a range.
- HListGroupLook - presentation of both the ListGroup itself and the items held on the list.
- HTextLook – a simple presentation mechanism for textual content.
- HSinglelineEntryLook – presentation of a single line of textual content that can be edited by the user.
- HMultilineEntryLook – presentation of multiple lines of textual content that can be edited by the user.

This basic set allows the construction of most typical interactive user interfaces when used in conjunction with the Component Behaviors to define a widget set. It can however be extended in a general fashion to provide new categories of "look".

When a widget is constructed, it is provided with a default look. This default HLook will be one of the standard set of looks listed above. For example, the HGraphicButton is created with the HGraphicLook by default. The default look that is used when the widget is constructed can be changed by calling the static method setDefaultLook that is provided on all widget types. Any widget of that type created after the call will be created with the new HLook that was passed in as the parameter to setDefaultLook. The look of an individual widget can be modified by using the method HVisible.setLook.

The Pluggable Look mechanism is flexible enough so that the application developer can create new HLooks. For example a combined HGraphicLook and HTextLook, where the Text may overlay the Graphic, or be shown in place of the Graphic while the Graphic is being loaded.

8.4.6 Content Behavior

Content is associated with the widget through the following methods on HVisible: setTextContent, setGraphicContent, setAnimateContent and setContent. Hence, multiple content (Text, Graphics, Animations and user-defined content) can be associated with a widget. The way multiple content is rendered is dependent on the HLook associated with the widget. The default looks provided by the platform may not render all the content types.

Different content can be associated with the different states of the widget. For example, an HGraphicButton might have six different images to represent its six different states (NORMAL_STATE, FOCUSED_STATE, ACTIONED_STATE, ACTIONED_FOCUSED_STATE, DISABLED_STATE and DISABLED_FOCUSED_STATE) to the user, using the setGraphicContent. The same content can be applied to all states of the widget by using the HState constant ALL_STATES when calling setTextContent, setGraphicContent, setAnimateContent and setContent.

- By default, content associated with a widget is not modified to fit the dimensions of the widget. Refer to the alignment and scaling methods in HVisible to address this issue.

Mechanisms are also available that allow (graphic) content to be resized to match the widget dimensions, etc.

8.5 HAVi Resident Widgets

Using the Component Behaviors (HVisible, HNavigable, HActionable and HSwitchable) defined in the previous section a set of resident widgets is provided.

Note that implementations of the HAVi widget set shall be implemented (and behave) as lightweight components. HAVi widgets do not include an associated peer class, irrespective of the exact mechanism for their implementation, i.e., directly implemented in Java, or via some platform specific mechanism.

8.5.1 Simple Text/Graphic/Animate Widgets

The HAVi set of resident widgets includes both visible and navigable versions of the HText, HIcon and HAnimation classes:

- Applications providing simple “display-only” non-navigable text, image, or animations may employ the “Static” versions of these classes.
- Applications wishing to provide additional feedback, e.g. “tooltips”, or audio feedback – for example, a commentary – may employ the navigable versions of these classes.

Widget Type	Description	Static	Navigable
Animation	Displays a simple sequence of images	HStaticAnimation	HAnimation
Text	Displays a text label	HStaticText	HText
Graphic	Displays an Image	HStaticIcon	HIcon

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.5.2 Buttons

The HAVi set of resident widgets includes both textual and graphical version of a push button: HTextButton and HGraphicButton. These buttons implement the HActionable interface that defines their behavior.

The HToggleButton is used to represent a graphical control that has a boolean state that can be toggled on and off by the user (e.g. Checkbox or Radio Button). The HToggleButton implements the HSwitchable interface that defines its behavior. A HToggleButton widget does not have an associated text label as part of the widget. If a text label is required, a separate HStaticText widget should be created.

A set of HToggleButtons can be associated with a HToggleGroup. A HToggleGroup will ensure that a maximum of one HToggleButton is chosen at any time (i.e. a group of radio buttons).

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.5.3 Range Widgets

The HAVi set of resident widgets include a group of controls to represent a particular integer value in a range of values i.e. a slider control, or scroll bar. The HStaticRange widget is a non navigable widget, HRange widget is navigable (implements the HNavigable interface) and the HRangeValue is

navigable and its value can be modified by user interaction (implements the HAdjustmentValue interface).

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.5.4 List Widgets

A HListGroup is a visible that manages a dynamic set of vertically or horizontally scrollable HListElements, allowing either single or multiple HListElements to be chosen by the user. The HListGroup will automatically scroll the HListElements when the user navigates to an element that is currently not visible within the list group.

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.5.5 Text Entry Widgets

The HSinglelineEntry component allows a user to enter a single line text string. A typical rendering is as a text entry field, e.g. with an associated on-screen keyboard. The HMultilineEntry widget extends the HSinglelineEntry widget and allows text to be entered over multiple lines. Both these widgets implement the HTextValue interface resulting in the widgets firing HTextEvents when starting to edit, finishing editing, and whenever the content changes.

Refer to the supporting Javadoc for a more detailed description of these widgets.

8.6 Profiles

Implementations of the HAVi User-Interface on an FAV should:

- have a minimum screen resolution of 320 by 240 pixels (quarter VGA)
- include support for the image and sound content types, as defined in DDI.
- either provide a physical keyboard or provide a virtual keyboard supporting at least the entry of alphanumeric codes

8.7 General Approach to Error Behavior

Where a method call is specified as taking an object as one of its input parameters, if null is passed as a parameter and not explicitly identified as a valid input for that parameter of that method then a java.lang.NullPointerException shall be thrown. Where an input parameter to a method call is specified to be more restrictive than its Java type allows (e.g. only a restricted set of numbers are allowed as inputs), providing values outside the allowed range shall result in a java.lang.IllegalArgumentException being thrown.

8.8 Register of Constants

public final static int org.havi.ui.HAdjustableLook.ADJUST_THUMB = -6;
public final static int org.havi.ui.HAdjustableLook.ADJUST_PAGE_MORE = -5;
public final static int org.havi.ui.HAdjustableLook.ADJUST_PAGE_LESS = -4;
public final static int org.havi.ui.HAdjustableLook.ADJUST_BUTTON_MORE = -3;
public final static int org.havi.ui.HAdjustableLook.ADJUST_BUTTON_LESS = -2;
public final static int org.havi.ui.HAdjustableLook.ADJUST_NONE = -1;

public final static int org.havi.ui.HAnimateEffect.REPEAT_INFINITE = -1;
public final static int org.havi.ui.HAnimateEffect.PLAY_REPEATING = 1;
public final static int org.havi.ui.HAnimateEffect.PLAY_ALTERNATING = 2;
public final static int org.havi.ui.HBackgroundConfigTemplate.CHANGEABLE_SINGLE_COLOR = 10;
public final static int org.havi.ui.HBackgroundConfigTemplate.STILL_IMAGE = 11;
public final static int org.havi.ui.HFontCapabilities.BASIC_LATIN = 1;
public final static int org.havi.ui.HFontCapabilities.LATIN_1_SUPPLEMENT = 2;
public final static int org.havi.ui.HFontCapabilities.LATIN_EXTENDED_A = 3;
public final static int org.havi.ui.HFontCapabilities.LATIN_EXTENDED_B = 4;
public final static int org.havi.ui.HFontCapabilities.IPA_EXTENSIONS = 5;
public final static int org.havi.ui.HFontCapabilities.SPACING_MODIFIER_LETTERS = 6;
public final static int org.havi.ui.HFontCapabilities.COMBINING_DIACRITICAL_MARKS = 7;
public final static int org.havi.ui.HFontCapabilities.BASIC_GREEK = 8;
public final static int org.havi.ui.HFontCapabilities.GREEK_SYMBOLS_AND_COPTIC = 9;
public final static int org.havi.ui.HFontCapabilities.CYRILLIC = 10;
public final static int org.havi.ui.HFontCapabilities.ARMENIAN = 11;
public final static int org.havi.ui.HFontCapabilities.BASIC_HEBREW = 12;
public final static int org.havi.ui.HFontCapabilities.HEBREW_EXTENDED = 13;
public final static int org.havi.ui.HFontCapabilities.BASIC_ARABIC = 14;
public final static int org.havi.ui.HFontCapabilities.ARABIC_EXTENDED = 15;
public final static int org.havi.ui.HFontCapabilities.DEVANAGARI = 16;
public final static int org.havi.ui.HFontCapabilities.BENGALI = 17;
public final static int org.havi.ui.HFontCapabilities.GURMUKHI = 18;
public final static int org.havi.ui.HFontCapabilities.GUJARATI = 19;
public final static int org.havi.ui.HFontCapabilities.ORIYA = 20;
public final static int org.havi.ui.HFontCapabilities.TAMIL = 21;
public final static int org.havi.ui.HFontCapabilities.TELUGU = 22;
public final static int org.havi.ui.HFontCapabilities.KANNADA = 23;
public final static int org.havi.ui.HFontCapabilities.MALAYALAM = 24;
public final static int org.havi.ui.HFontCapabilities.THAI = 25;
public final static int org.havi.ui.HFontCapabilities.LAO = 26;
public final static int org.havi.ui.HFontCapabilities.BASIC_GEORGIAN = 27;
public final static int org.havi.ui.HFontCapabilities.GEORGIAN_EXTENDED = 28;
public final static int org.havi.ui.HFontCapabilities.HANGUL_JAMO = 29;
public final static int org.havi.ui.HFontCapabilities.LATIN_EXTENDED_ADDITIONAL = 30;
public final static int org.havi.ui.HFontCapabilities.GREEK_EXTENDED = 31;
public final static int org.havi.ui.HFontCapabilities.GENERAL_PUNCTUATION = 32;
public final static int org.havi.ui.HFontCapabilities.SUPERSCRIPTS_AND_SUBSCRIPTS = 33;
public final static int org.havi.ui.HFontCapabilities.CURRENCY_SYMBOLS = 34;
public final static int org.havi.ui.HFontCapabilities.COMBINING_DIACTRICAL_MARKS_FOR_SYMBOLS =
35;
public final static int org.havi.ui.HFontCapabilities.LETTERLIKE_SYMBOLS = 36;
public final static int org.havi.ui.HFontCapabilities.NUMBER_FORMS = 37;
public final static int org.havi.ui.HFontCapabilities.ARROWS = 38;
public final static int org.havi.ui.HFontCapabilities.MATHEMATICAL_OPERATORS = 39;
public final static int org.havi.ui.HFontCapabilities.MISCELLANEOUS_TECHNICAL = 40;
public final static int org.havi.ui.HFontCapabilities.CONTROL_PICTURES = 41;
public final static int org.havi.ui.HFontCapabilities.OPTICAL_CHARACTER_RECOGNITION = 42;
public final static int org.havi.ui.HFontCapabilities.ENCLOSED_ALPHANUMERICS = 43;
public final static int org.havi.ui.HFontCapabilities.BOX_DRAWING = 44;
public final static int org.havi.ui.HFontCapabilities.BLOCK_ELEMENTS = 45;
public final static int org.havi.ui.HFontCapabilities.GEOMETRICAL_SHAPES = 46;
public final static int org.havi.ui.HFontCapabilities.MISCELLANEOUS_SYMBOLS = 47;
public final static int org.havi.ui.HFontCapabilities.DINGBATS = 48;
public final static int org.havi.ui.HFontCapabilities.CJK_SYMBOLS_AND_PUNCTUATION = 49;
public final static int org.havi.ui.HFontCapabilities.HIRAGANA = 50;
public final static int org.havi.ui.HFontCapabilities.KATAKANA = 51;
public final static int org.havi.ui.HFontCapabilities.BOPOMOFO = 52;
public final static int org.havi.ui.HFontCapabilities.HANGUL_COMPATIBILITY_JAMO = 53;
public final static int org.havi.ui.HFontCapabilities.CJK_MISCELLANEOUS = 54;
public final static int org.havi.ui.HFontCapabilities.ENCLOSED_CJK_LETTERS_AND_MONTHS = 55;
public final static int org.havi.ui.HFontCapabilities.CJK_COMPATIBILITY = 56;
public final static int org.havi.ui.HFontCapabilities.HANGUL = 57;
public final static int org.havi.ui.HFontCapabilities.HANGUL_SUPPLEMENTARY_A = 58;
public final static int org.havi.ui.HFontCapabilities.HANGUL_SUPPLEMENTARY_B = 59;
public final static int org.havi.ui.HFontCapabilities.CJK_UNIFIED_IDEOGRAPHS = 60;
public final static int org.havi.ui.HFontCapabilities.PRIVATE_USE_AREA = 61;

```
public final static int org.havi.ui.HFontCapabilities.CJK_COMPATIBILITY_IDEOGRAPHS = 62;
public final static int org.havi.ui.HFontCapabilities.ALPHABETIC_PRESENTATION_FORMS_A = 63;
public final static int org.havi.ui.HFontCapabilities.ARABIC_PRESENTATION_FORMS_A = 64;
public final static int org.havi.ui.HFontCapabilities.COMBINING_HALF_MARKS = 65;
public final static int org.havi.ui.HFontCapabilities.CJK_COMPATIBILITY_FORMS = 66;
public final static int org.havi.ui.HFontCapabilities.SMALL_FORM_VARIANTS = 67;
public final static int org.havi.ui.HFontCapabilities.ARABIC_PRESENTATION_FORMS_B = 68;
public final static int org.havi.ui.HFontCapabilities.HALFWIDTH_AND_FULLWIDTH_FORMS = 69;
public final static int org.havi.ui.HFontCapabilities.SPECIALS = 70;
public final static int org.havi.ui.HGraphicsConfigTemplate.VIDEO_MIXING = 12;
public final static int org.havi.ui.HGraphicsConfigTemplate.MATTE_SUPPORT = 13;
public final static int org.havi.ui.HGraphicsConfigTemplate.IMAGE_SCALING_SUPPORT = 14;
public final static int org.havi.ui.HImageHints.NATURAL_IMAGE = 1;
public final static int org.havi.ui.HImageHints.CARTOON = 2;
public final static int org.havi.ui.HImageHints.BUSINESS_GRAPHICS = 3;
public final static int org.havi.ui.HImageHints.LINE_ART = 4;
public final static int org.havi.ui.HKeyboardInputPreferred.INPUT_NUMERIC = 1;
public final static int org.havi.ui.HKeyboardInputPreferred.INPUT_ALPHA = 2;
public final static int org.havi.ui.HKeyboardInputPreferred.INPUT_ANY = 4;
public final static int org.havi.ui.HKeyboardInputPreferred.INPUT_CUSTOMIZED = 8;
public final static int org.havi.ui.HListGroup.DEFAULT_ICON_HEIGHT = -4;
public final static int org.havi.ui.HListGroup.DEFAULT_ICON_WIDTH = -3;
public final static int org.havi.ui.HListGroup.DEFAULT_LABEL_HEIGHT = -2;
public final static int org.havi.ui.HListGroup.ITEM_NOT_FOUND = -1;
public final static int org.havi.ui.HListGroup.ADD_INDEX_END = -1;
public final static int org.havi.ui.HListGroup.DEFAULT_LABEL_WIDTH = -1;
public final static int org.havi.ui.HOrientable.ORIENT_LEFT_TO_RIGHT = 0;
public final static int org.havi.ui.HOrientable.ORIENT_RIGHT_TO_LEFT = 1;
public final static int org.havi.ui.HOrientable.ORIENT_TOP_TO_BOTTOM = 2;
public final static int org.havi.ui.HOrientable.ORIENT_BOTTOM_TO_TOP = 3;
public final static int org.havi.ui.HScene.IMAGE_NONE = 0;
public final static int org.havi.ui.HScene.NO_BACKGROUND_FILL = 0;
public final static int org.havi.ui.HScene.IMAGE_STRETCH = 1;
public final static int org.havi.ui.HScene.BACKGROUND_FILL = 1;
public final static int org.havi.ui.HScene.IMAGE_CENTER = 2;
public final static int org.havi.ui.HScene.IMAGE_TILE = 3;
public final static int org.havi.ui.HSceneTemplate.GRAPHICS_CONFIGURATION = 0;
public final static int org.havi.ui.HSceneTemplate.REQUIRED = 1;
public final static int org.havi.ui.HSceneTemplate.SCENE_PIXEL_DIMENSION = 1;
public final static int org.havi.ui.HSceneTemplate.PREFERRED = 2;
public final static int org.havi.ui.HSceneTemplate.SCENE_PIXEL_LOCATION = 2;
public final static int org.havi.ui.HSceneTemplate.UNNECESSARY = 3;
public final static int org.havi.ui.HSceneTemplate.SCENE_SCREEN_DIMENSION = 4;
public final static int org.havi.ui.HSceneTemplate.SCENE_SCREEN_LOCATION = 8;
public final static int org.havi.ui.HScreenConfigTemplate.REQUIRED = 1;
public final static int org.havi.ui.HScreenConfigTemplate.ZERO_BACKGROUND_IMPACT = 1;
public final static int org.havi.ui.HScreenConfigTemplate.PREFERRED = 2;
public final static int org.havi.ui.HScreenConfigTemplate.ZERO_GRAPHICS_IMPACT = 2;
public final static int org.havi.ui.HScreenConfigTemplate.DONT_CARE = 3;
public final static int org.havi.ui.HScreenConfigTemplate.ZERO_VIDEO_IMPACT = 3;
public final static int org.havi.ui.HScreenConfigTemplate.PREFERRED_NOT = 4;
public final static int org.havi.ui.HScreenConfigTemplate.INTERLACED_DISPLAY = 4;
public final static int org.havi.ui.HScreenConfigTemplate.REQUIRED_NOT = 5;
public final static int org.havi.ui.HScreenConfigTemplate.FLICKER_FILTERING = 5;
public final static int org.havi.ui.HScreenConfigTemplate.VIDEO_GRAPHICS_PIXEL_ALIGNED = 6;
public final static int org.havi.ui.HScreenConfigTemplate.PIXEL_ASPECT_RATIO = 7;
public final static int org.havi.ui.HScreenConfigTemplate.PIXEL_RESOLUTION = 8;
public final static int org.havi.ui.HScreenConfigTemplate.SCREEN_RECTANGLE = 9;
public final static int org.havi.ui.HState.FOCUSED_STATE_BIT = 1;
public final static int org.havi.ui.HState.ACTIONED_STATE_BIT = 2;
public final static int org.havi.ui.HState.DISABLED_STATE_BIT = 4;
public final static int org.havi.ui.HState.ALL_STATES = 7;
public final static int org.havi.ui.HState.FIRST_STATE = 128;
public final static int org.havi.ui.HState.NORMAL_STATE = 128;
public final static int org.havi.ui.HState.FOCUSED_STATE = 129;
public final static int org.havi.ui.HState.ACTIONED_STATE = 130;
public final static int org.havi.ui.HState.ACTIONED_FOCUSED_STATE = 131;
```

public final static int org.havi.ui.HState.DISABLED_STATE = 132;
public final static int org.havi.ui.HState.DISABLED_FOCUSED_STATE = 133;
public final static int org.havi.ui.HState.DISABLED_ACTIONED_STATE = 134;
public final static int org.havi.ui.HState.DISABLED_ACTIONED_FOCUSED_STATE = 135;
public final static int org.havi.ui.HState.LAST_STATE = 135;
public final static int org.havi.ui.HStaticRange.SLIDER_BEHAVIOR = 0;
public final static int org.havi.ui.HStaticRange.SCROLLBAR_BEHAVIOR = 1;
public final static int org.havi.ui.HVideoConfigTemplate.GRAPHICS_MIXING = 15;
public final static int org.havi.ui.HVisible.NO_DEFAULT_WIDTH = -1;
public final static int org.havi.ui.HVisible.NO_DEFAULT_HEIGHT = -1;
public final static int org.havi.ui.HVisible.HALIGN_LEFT = 0;
public final static int org.havi.ui.HVisible.VALIGN_TOP = 0;
public final static int org.havi.ui.HVisible.RESIZE_NONE = 0;
public final static int org.havi.ui.HVisible.NO_BACKGROUND_FILL = 0;
public final static int org.havi.ui.HVisible.FIRST_CHANGE = 0;
public final static int org.havi.ui.HVisible.TEXT_CONTENT_CHANGE = 0;
public final static int org.havi.ui.HVisible.HALIGN_CENTER = 1;
public final static int org.havi.ui.HVisible.RESIZE_PRESERVE_ASPECT = 1;
public final static int org.havi.ui.HVisible.BACKGROUND_FILL = 1;
public final static int org.havi.ui.HVisible.GRAPHIC_CONTENT_CHANGE = 1;
public final static int org.havi.ui.HVisible.HALIGN_RIGHT = 2;
public final static int org.havi.ui.HVisible.RESIZE_ARBITRARY = 2;
public final static int org.havi.ui.HVisible.ANIMATE_CONTENT_CHANGE = 2;
public final static int org.havi.ui.HVisible.HALIGN_JUSTIFY = 3;
public final static int org.havi.ui.HVisible.CONTENT_CHANGE = 3;
public final static int org.havi.ui.HVisible.VALIGN_CENTER = 4;
public final static int org.havi.ui.HVisible.STATE_CHANGE = 4;
public final static int org.havi.ui.HVisible.CARET_POSITION_CHANGE = 5;
public final static int org.havi.ui.HVisible.ECHO_CHAR_CHANGE = 6;
public final static int org.havi.ui.HVisible.EDIT_MODE_CHANGE = 7;
public final static int org.havi.ui.HVisible.VALIGN_BOTTOM = 8;
public final static int org.havi.ui.HVisible.MIN_MAX_CHANGE = 8;
public final static int org.havi.ui.HVisible.THUMB_OFFSETS_CHANGE = 9;
public final static int org.havi.ui.HVisible.ORIENTATION_CHANGE = 10;
public final static int org.havi.ui.HVisible.TEXT_VALUE_CHANGE = 11;
public final static int org.havi.ui.HVisible.VALIGN_JUSTIFY = 12;
public final static int org.havi.ui.HVisible.ITEM_VALUE_CHANGE = 12;
public final static int org.havi.ui.HVisible.ADJUSTMENT_VALUE_CHANGE = 13;
public final static int org.havi.ui.HVisible.LIST_CONTENT_CHANGE = 14;
public final static int org.havi.ui.HVisible.LIST_ICONSIZE_CHANGE = 15;
public final static int org.havi.ui.HVisible.LIST_LABELSIZE_CHANGE = 16;
public final static int org.havi.ui.HVisible.LIST_MULTISELECTION_CHANGE = 17;
public final static int org.havi.ui.HVisible.LIST_SCROLLPOSITION_CHANGE = 18;
public final static int org.havi.ui.HVisible.SIZE_CHANGE = 19;
public final static int org.havi.ui.HVisible.BORDER_CHANGE = 20;
public final static int org.havi.ui.HVisible.REPEAT_COUNT_CHANGE = 21;
public final static int org.havi.ui.HVisible.ANIMATION_POSITION_CHANGE = 22;
public final static int org.havi.ui.HVisible.LIST_SELECTION_CHANGE = 23;
public final static int org.havi.ui.HVisible.UNKNOWN_CHANGE = 24;
public final static int org.havi.ui.HVisible.LAST_CHANGE = 24;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_FIRST = 2000;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_START_CHANGE = 2000;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_LESS = 2001;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_MORE = 2002;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_PAGE_LESS = 2003;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_PAGE_MORE = 2004;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_LAST = 2005;
public final static int org.havi.ui.event.HAdjustmentEvent.ADJUST_END_CHANGE = 2005;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_FIRST = 1;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_LOADED = 1;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_FILE_NOT_FOUND
= 2;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_IOERROR = 3;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_INVALID = 4;
public final static int org.havi.ui.event.HBackgroundImageEvent.BACKGROUNDIMAGE_LAST = 4;
public final static int org.havi.ui.event.HEventRepresentation.ER_TYPE_NOT_SUPPORTED = 0;
public final static int org.havi.ui.event.HEventRepresentation.ER_TYPE_STRING = 1;

public final static int org.havi.ui.event.HEventRepresentation.ER_TYPE_COLOR = 2;
public final static int org.havi.ui.event.HEventRepresentation.ER_TYPE_SYMBOL = 4;
public final static int org.havi.ui.event.HFocusEvent.NO_TRANSFER_ID = -1;
public final static int org.havi.ui.event.HFocusEvent.HFOCUS_FIRST = 2029;
public final static int org.havi.ui.event.HFocusEvent.FOCUS_TRANSFER = 2029;
public final static int org.havi.ui.event.HFocusEvent.HFOCUS_LAST = 2029;
public final static int org.havi.ui.event.HItemEvent.ITEM_FIRST = 2006;
public final static int org.havi.ui.event.HItemEvent.ITEM_START_CHANGE = 2006;
public final static int org.havi.ui.event.HItemEvent.ITEM_TOGGLE_SELECTED = 2007;
public final static int org.havi.ui.event.HItemEvent.ITEM_SELECTED = 2008;
public final static int org.havi.ui.event.HItemEvent.ITEM_CLEARED = 2009;
public final static int org.havi.ui.event.HItemEvent.ITEM_SELECTION_CLEARED = 2010;
public final static int org.havi.ui.event.HItemEvent.ITEM_SET_CURRENT = 2011;
public final static int org.havi.ui.event.HItemEvent.ITEM_SET_PREVIOUS = 2012;
public final static int org.havi.ui.event.HItemEvent.ITEM_SET_NEXT = 2013;
public final static int org.havi.ui.event.HItemEvent.SCROLL_MORE = 2014;
public final static int org.havi.ui.event.HItemEvent.SCROLL_LESS = 2015;
public final static int org.havi.ui.event.HItemEvent.SCROLL_PAGE_MORE = 2016;
public final static int org.havi.ui.event.HItemEvent.SCROLL_PAGE_LESS = 2017;
public final static int org.havi.ui.event.HItemEvent.ITEM_END_CHANGE = 2018;
public final static int org.havi.ui.event.HItemEvent.ITEM_LAST = 2018;
public final static int org.havi.ui.event.HRcEvent.RC_FIRST = 400;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_0 = 403;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_1 = 404;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_2 = 405;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_3 = 406;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_4 = 407;
public final static int org.havi.ui.event.HRcEvent.VK_COLORED_KEY_5 = 408;
public final static int org.havi.ui.event.HRcEvent.VK_POWER = 409;
public final static int org.havi.ui.event.HRcEvent.VK_DIMMER = 410;
public final static int org.havi.ui.event.HRcEvent.VK_WINK = 411;
public final static int org.havi.ui.event.HRcEvent.VK_REWIND = 412;
public final static int org.havi.ui.event.HRcEvent.VK_STOP = 413;
public final static int org.havi.ui.event.HRcEvent.VK_EJECT_TOGGLE = 414;
public final static int org.havi.ui.event.HRcEvent.VK_PLAY = 415;
public final static int org.havi.ui.event.HRcEvent.VK_RECORD = 416;
public final static int org.havi.ui.event.HRcEvent.VK_FAST_FWD = 417;
public final static int org.havi.ui.event.HRcEvent.VK_PLAY_SPEED_UP = 418;
public final static int org.havi.ui.event.HRcEvent.VK_PLAY_SPEED_DOWN = 419;
public final static int org.havi.ui.event.HRcEvent.VK_PLAY_SPEED_RESET = 420;
public final static int org.havi.ui.event.HRcEvent.VK_RECORD_SPEED_NEXT = 421;
public final static int org.havi.ui.event.HRcEvent.VK_GO_TO_START = 422;
public final static int org.havi.ui.event.HRcEvent.VK_GO_TO_END = 423;
public final static int org.havi.ui.event.HRcEvent.VK_TRACK_PREV = 424;
public final static int org.havi.ui.event.HRcEvent.VK_TRACK_NEXT = 425;
public final static int org.havi.ui.event.HRcEvent.VK_RANDOM_TOGGLE = 426;
public final static int org.havi.ui.event.HRcEvent.VK_CHANNEL_UP = 427;
public final static int org.havi.ui.event.HRcEvent.VK_CHANNEL_DOWN = 428;
public final static int org.havi.ui.event.HRcEvent.VK_STORE_FAVORITE_0 = 429;
public final static int org.havi.ui.event.HRcEvent.VK_STORE_FAVORITE_1 = 430;
public final static int org.havi.ui.event.HRcEvent.VK_STORE_FAVORITE_2 = 431;
public final static int org.havi.ui.event.HRcEvent.VK_STORE_FAVORITE_3 = 432;
public final static int org.havi.ui.event.HRcEvent.VK_RECALL_FAVORITE_0 = 433;
public final static int org.havi.ui.event.HRcEvent.VK_RECALL_FAVORITE_1 = 434;
public final static int org.havi.ui.event.HRcEvent.VK_RECALL_FAVORITE_2 = 435;
public final static int org.havi.ui.event.HRcEvent.VK_RECALL_FAVORITE_3 = 436;
public final static int org.havi.ui.event.HRcEvent.VK_CLEAR_FAVORITE_0 = 437;
public final static int org.havi.ui.event.HRcEvent.VK_CLEAR_FAVORITE_1 = 438;
public final static int org.havi.ui.event.HRcEvent.VK_CLEAR_FAVORITE_2 = 439;
public final static int org.havi.ui.event.HRcEvent.VK_CLEAR_FAVORITE_3 = 440;
public final static int org.havi.ui.event.HRcEvent.VK_SCAN_CHANNELS_TOGGLE = 441;
public final static int org.havi.ui.event.HRcEvent.VK_PINP_TOGGLE = 442;
public final static int org.havi.ui.event.HRcEvent.VK_SPLIT_SCREEN_TOGGLE = 443;
public final static int org.havi.ui.event.HRcEvent.VK_DISPLAY_SWAP = 444;
public final static int org.havi.ui.event.HRcEvent.VK_SCREEN_MODE_NEXT = 445;
public final static int org.havi.ui.event.HRcEvent.VK_VIDEO_MODE_NEXT = 446;
public final static int org.havi.ui.event.HRcEvent.VK_VOLUME_UP = 447;

public final static int org.havi.ui.event.HRcEvent.VK_VOLUME_DOWN = 448;
public final static int org.havi.ui.event.HRcEvent.VK_MUTE = 449;
public final static int org.havi.ui.event.HRcEvent.VK_SURROUND_MODE_NEXT = 450;
public final static int org.havi.ui.event.HRcEvent.VK_BALANCE_RIGHT = 451;
public final static int org.havi.ui.event.HRcEvent.VK_BALANCE_LEFT = 452;
public final static int org.havi.ui.event.HRcEvent.VK_FADER_FRONT = 453;
public final static int org.havi.ui.event.HRcEvent.VK_FADER_REAR = 454;
public final static int org.havi.ui.event.HRcEvent.VK_BASS_BOOST_UP = 455;
public final static int org.havi.ui.event.HRcEvent.VK_BASS_BOOST_DOWN = 456;
public final static int org.havi.ui.event.HRcEvent.VK_INFO = 457;
public final static int org.havi.ui.event.HRcEvent.VK_GUIDE = 458;
public final static int org.havi.ui.event.HRcEvent.VK_TELETEXT = 459;
public final static int org.havi.ui.event.HRcEvent.VK_SUBTITLE = 460;
public final static int org.havi.ui.event.HRcEvent.RC_LAST = 460;
public final static int org.havi.ui.event.HTextEvent.TEXT_FIRST = 2019;
public final static int org.havi.ui.event.HTextEvent.TEXT_START_CHANGE = 2019;
public final static int org.havi.ui.event.HTextEvent.TEXT_CHANGE = 2020;
public final static int org.havi.ui.event.HTextEvent.TEXT_CARET_CHANGE = 2021;
public final static int org.havi.ui.event.HTextEvent.TEXT_END_CHANGE = 2022;
public final static int org.havi.ui.event.HTextEvent.CARET_NEXT_CHAR = 2023;
public final static int org.havi.ui.event.HTextEvent.CARET_NEXT_LINE = 2024;
public final static int org.havi.ui.event.HTextEvent.CARET_PREV_CHAR = 2025;
public final static int org.havi.ui.event.HTextEvent.CARET_PREV_LINE = 2026;
public final static int org.havi.ui.event.HTextEvent.CARET_NEXT_PAGE = 2027;
public final static int org.havi.ui.event.HTextEvent.TEXT_LAST = 2028;
public final static int org.havi.ui.event.HTextEvent.CARET_PREV_PAGE = 2028;

9 SDD Data

9.1 References

At the beginning of this document is a collection of references. Each of the IEEE 1212 ROM fields defined in this section includes the reference indicator, either [1], [2] or [4] for convenience to the reader. Fields which are defined in this document specifically for the HAVi architecture will be appended with [HAVi] as the reference indicator.

9.2 Introduction

This section describes the SDD data for HAVi-compliant 1394 devices. This information is stored in the configuration ROM of the device, according to the layout rules defined by [1] and [2].

About the IEEE 1212 specification: the current official draft is [1]. However, the specification is currently undergoing a revision process as part of its 5-year re-evaluation. Some of the newly defined items for this revision are necessary for the HAVi architecture. The newly defined data structures are in [2]. Additional HAVi-specific data structures are defined in this document.

9.3 Text Encoding Formats

1212 reference [1] defines only a minimal English ASCII text encoding format. The 1212 revision [2] defines several additional 2-byte formats, including UNICODE. The text fields of SDD use UNICODE encoding.

9.4 HAVi Key Values

Each of the keys in the following table is defined within the scope of the HAVi Unit Directory. The definitions for each of these keys are presented in subsequent sections of this document.

Table 17. HAVi Unit Directory Key Values

HAVi Key Name	Key ID	Valid Key Types
HAVi_Device_Profile_Key	38_{16}	Immediate
HAVi_DCM_Key	39_{16}	Leaf offset
HAVi_DCM_Reference_Key	$3A_{16}$	Leaf offset
HAVi_DCM_Profile_Key	$3B_{16}$	Leaf offset
HAVi_Device_Icon_Bitmap_Key	$3C_{16}$	Leaf offset
HAVi_Message_Version_Key	$3D_{16}$	Immediate

9.5 Minimum Required Data

The fields described in this document are the minimum required for HAVi-compliant devices, but these may not be all that is necessary for the device being implemented. Depending on other standards supported by the device, additional fields may be required.

If a device changes any of the HAVi defined fields, except the user preferred name, it shall generate a 1394 bus reset. Moreover, a HAVi controller must also generate a bus reset after initialization or self-test, i.e., when the first quadlet of its configuration ROM transits from a zero to non-zero value. This bus reset is necessary to re-detect HAVi controller devices and restart HAVi specific protocols between controllers.

The following table illustrates the required and optional HAVi-specific data:

Table 18. HAVi Configuration ROM Requirements

HAVi Key Name	Requirement
HAVi_Device_Profile_Key	Required for all HAVi-compliant device classes
modifiable descriptor for HAVi_Device_Profile	Required for all HAVi-compliant device classes
HAVi_DCM_Key	Strongly recommended for BAV Does not apply for IAV, FAV
HAVi_DCM_Reference_Key	Strongly recommended for BAV Does not apply for IAV, FAV
HAVi_DCM_Profile_Key	Required if DCM is included (BAV) Does not apply for IAV, FAV
HAVi_Device_Icon_Bitmap_Key	Optional for all HAVi-compliant device classes
HAVi_Message_Version_Key	Required for IAV, FAV, does not apply for BAV

The following table illustrates the required non-HAVi-specific data (defined by other standards, such as [1] and [2]):

Table 19. Non-HAVi Configuration ROM Requirements

Key_ID	Key Name	Requirement	Directory	Described in [2]
18_{16}	**Instance_Directory**	Required for all HAVi device classes	Root directory	7.7.15
17_{16}	**Model_ID**	Required for all HAVi device classes	Root directory (exceptionally vendor or instance directory)	7.7.14
11_{16}	**HAVi_Unit_Directory**	Required for all HAVi device classes	Instance directory (and possibly root)	7.7.9
11_{16}	**IEC_61883_Unit_Directory**	Required for HAVi devices supporting multiple 61883 protocols	Root and instance directory	7.7.9
12_{16}	**Specifier_ID**	Required for all HAVi device classes	HAVi unit directory	7.7.10
13_{16}	**Version**	Required for all HAVi device classes	HAVi unit directory	7.7.11

9.6 ROM Format

Devices shall implement the general ROM format as defined in [1].

9.7 The GUID and the Bus_Info_Block

In the **Bus_Info_Block** of the configuration ROM, the 8 bit chip_id_hi field is concatenated with the 32 bit chip_id_lo to create a 40 bit chip ID value. The vendor specified by the node_vendor_id value shall administer the chip ID value. When appended to the node_vendor_id value, these shall form a unique 64-bit value called **EUI-64** (Extended Unique Identifier, 64 bits). The HAVi GUID (Global Unique Identifier) reflects that **EUI-64** value.

9.8 Root Directory

Devices shall implement a root directory with the following fields:

9.8.1 Vendor_ID [2]

The **Vendor_ID** is a 24-bit immediate value (registration ID), assigned by the IEEE, which is globally unique. This value identifies the vendor that manufactured the device.

9.8.2 HAVi_Unit_Directory [1]

1394 devices conformant with the HAVi specification are required to support the IEC 61883 standard for data transmission formats. All devices shall implement a **Unit_Directory** field which points to a 1212 Unit Directory which specifies the IEC 61883 protocol and the HAVi protocol. This structure is described in more detail below.

The **HAVi_Unit_Directory** entry is mandatory in the instance directory. A second entry in the root directory is present only to allow older devices compatible with the current version of the IEC 61883 standard to find this unit directory. If present, the **HAVi_Unit_Directory** entry shall be the only unit directory entry in the root, i.e., the **HAVi_Unit_Directory** entry will be present in the root only if no other **IEC_61883_Unit_Directory** (see section 9.8.3) is present in the root. Since the p1212 revision [2] obsoletes this entry, HAVi devices shall rely only on the **HAVi_Unit_Directory** entry of the instance directory.

9.8.3 Other IEC_61883_Unit_Directory [1] [4]

HAVi devices may optionally implement further protocols specified by the IEC 61883 CTS code. If this is the case, a separate unit directory is needed for those non-HAVi protocols.

9.8.4 Instance_Directory [2]

The newly defined **Instance_Directory** is an offset reference to a directory structure. An instance directory is mandatory for every HAVi device, since it contains the link for the HAVi Unit Directory.

9.8.5 Model_ID [2]

The **Model_ID** field is a binary value which identifies the device model. The format of this field is 24-bit immediate. The contents of the **Model_ID** are defined by the device vendor. The **Model_ID** value should represent a family or class of products and should not be unique to individual devices.

HAVi devices shall, if possible, have the **Model_ID** entry in the root directory. If this is not possible,

e.g., for multi-standard devices, the **Model_ID** may exceptionally be present in a root dependent vendor directory or the instance directory that contains the HAVi Unit Directory. If a **Model_ID** is present in both (and not in the root directory), the entries (and dependent descriptors) shall be identical. If a **Model_ID** entry is present in the root and elsewhere, the entries (and dependent descriptors) should be identical as HAVi considers only the root directory.

9.9 Instance Directory

Devices shall implement an **Instance_Directory** with the following fields:

9.9.1 HAVi_Unit_Directory [1][2]

This entry points to the same **HAVi_Unit_Directory** (see below). A second entry pointing to the **HAVi_Unit_Directory** may under certain conditions be present in the root directory. According to [2] HAVi devices shall rely only on the **HAVi_Unit_Directory** entry in the instance directory, the entry in the root directory is needed only for backwards compatibility.

9.10 HAVi Unit Directory

Devices shall declare a unit architecture with the following fields as a minimum:

9.10.1 Specifier_ID [1]

The **Specifier_ID** shall be the first entry of the Unit Directory.

The **Specifier_ID** in the HAVi Unit Directory is a 24-bit immediate value, and shall be that of the 1394 Trade Association, as follows:

first octet	00_{16}
second octet	$A0_{16}$
third octet	$2D_{16}$

9.10.2 Version [1]

The **Version** entry shall be the second entry in the Unit Directory.

In the **Version** field the two least significant bytes specify the version of the SDD fields which are defined in this device, as described below:

first octet	01_{16}
second octet	00_{16}
third octet	08_{16}

The combination of **Specifier_ID** and **Version** defines the meaning of all subsequent keys in the range of 38_{16} to $3F_{16}$ inclusive.

9.10.3 HAVi_Message_Version [HAVi]

The **HAVi_Message_Version** is a 24-bit immediate value, which specifies the version of the HAVi Messaging System supported by this device. It has the following definition:

first octet	Reserved (currently 00_{16})
second octet	Major Version Number (currently 01_{16})
third octet	Minor Version Number (currently $0A_{16}$)

9.10.4 HAVi_Device_Profile [HAVi]

The **HAVi_Device_Profile** is a 24-bit immediate value specifying the main capabilities of the device. The value is interpreted as a field of 24 bits with individual meanings, in the following description bits are numbered from Bit 0 for the LSB to Bit 23 for the MSB.

9.10.4.1 HAVi_Device_Class [Bit0..3]

The **HAVi_Device_Class** is a 4-bit immediate value specifying which of the non-LAV device categories this device conforms to. The following values are defined:

HAVi_Device_Class value	Meaning
0000_2	reserved
0001_2	BAV Device
0010_2	IAV Device
0011_2	FAV Device
other values	reserved

9.10.4.2 HAVi_DCM_Manager [Bit4]

The **HAVi_DCM_Manager** is a 1-bit immediate value specifying for IAV devices whether a DCM Manager is implemented. For a BAV this bit shall be 0, for a FAV this bit shall be 1.

HAVi_DCM_Manager value	Meaning
0	DCM Manager absent
1	DCM Manager present

9.10.4.3 HAVi_Stream_Manager [Bit5]

The **HAVi_Stream_Manager** is a 1-bit immediate value specifying for IAV devices whether a Stream Manager is implemented. For a BAV this bit shall be 0, for a FAV this bit shall be 1.

HAVi_Stream_Manager value	Meaning
0	Stream Manager absent
1	Stream Manager present

9.10.4.4 HAVi_Resource_Manager [Bit6]

The **HAVi_Resource_Manager** is a 1-bit immediate value specifying for IAV devices whether a Resource Manager is implemented. For a BAV this bit shall be 0, for a FAV this bit shall be 1.

HAVi_Resource_Manager value	Meaning
0	Resource Manager absent

1	Resource Manager present

9.10.4.5 HAVi_Display_Capability [Bit7]

The **HAVi_Display_Capability** is a 1-bit immediate value specifying for IAV devices whether a DDI Controller is implemented, and for FAV devices whether a DDI Controller and a Level 2 User Interface capability is implemented. For a BAV this bit shall be 0.

HAVi_Display_Capability value	Meaning
0	does not have display capability
1	has display capability

9.10.4.6 HAVi_Device_Status [Bit8]

This bit specifies the status of the device. For an IAV or FAV, a value of one indicates that the HAVi Messaging System and other system components are running – the device is prepared to receive and process incoming HAVi messages. For a BAV, a value of one indicates that the device is ready to accept native commands (e.g., AV/C) over 1394.

HAVi_Device_Status value	Meaning
0	inactive
1	active

9.10.4.7 Reserved Bits [Bit9..23]

The upper 15 bits of the **HAVi_Device_Profile** are reserved and shall read as 0.

9.10.5 HAVi_User_Preferred_Name [2][HAVi]

The **HAVi_User_Preferred_Name** entry is mandatory for all devices. It is coded as a single modifiable textual descriptor to the **HAVi_Device_Profile**. The device shall allow this textual descriptor to be modified and should store it persistently. When it is modified, the DCM must update the HAVi Registry. The means by which the device informs the DCM of an update (polling, notification …) is implementation dependent.

From HAVi devices, this field is modified by `Dcm::SetUserPreferredName`. The DCM then writes the user preferred name into the node's address space, while respecting the rules for modifiable descriptors. Non-HAVi controllers shall directly modify the user preferred name in the node's address space, while respecting the rules for modifiable descriptors. Both HAVi and non-HAVi controllers shall ensure that Width and Character Set are always set to fixed two byte unicode characters, the language field shall be set to all 0, to indicate "undefined". Note that the parameter leaf referenced by the **HAVi_User_Preferred_Name** entry contains a fixed max_descriptor_size value of 11, to allow values of up to 16 2-byte Unicode characters.

In multi-standard devices, nicknames in addition to the **HAVi_User_Preferred_Name** may exist (e.g., textual descriptors to the **Model_ID**). HAVi-aware controllers may in this case update all names in a coherent manner, using the same text with the appropriate character set. But other controllers may not be HAVi aware, and may update the "foreign" nickname without modifying the **HAVi_User_Preferred_Name**. Furthermore, some HAVi controllers may update only the **HAVi_User_Preferred_Name**. To avoid inconsistency it is expected that in such cases the device itself attempts updating its unmodified names.

9.10.6 HAVi_DCM [HAVi]

The **HAVi_DCM** is a 24-bit offset to a 1212 leaf structure. The leaf contains the DCM bytecode unit. DCM code units shall be encoded as described in section 7.4.1. This field and the leaf structure are not applicable for the IAV and FAV device classes; they are recommended for the BAV device class.

9.10.7 HAVi_DCM_Profile [HAVi]

The **HAVi_DCM_Profile** is a leaf structure which contains information about the DCM. It is structured as follows (the following diagram represents a quadlet-aligned structure with the least significant bytes on the right):

Figure 44. HAVi_DCM_Profile Leaf Structure

Leaf length	CRC
Transferred_DCM_Code_Unit_Size	
Installed_DCM_Code_Space	
Installed_DCM_Working_Space	
reserved	Message_Version

The **Transferred_DCM_Code_Unit_Size** field specifies the size, in bytes, of the DCM code unit as it would be transferred from source to destination (i.e., JAR file size). The **Installed_DCM_Code_Space** designates the required memory size of the installed code unit part (read-only), not including the amount of working space the code unit requires. The **Installed_DCM_Working_Space** is an estimate of the working space (read/write) the code unit requires. Note that an installed DCM code unit includes installed DCM and FCMs embedded in the code unit.

The **Message_Version** field specifies the lowest version of the HAVi Messaging System required by this DCM. This field is interpreted as defined in the HAVi Messaging System description.

9.10.8 HAVi_DCM_Reference [HAVi]

The **HAVi_DCM_Reference** is a URL which provides remote access to a BAV DCM code unit and its profile. This field is implemented as a leaf structure. It is a free-standing text string which contains the URL. The **HAVi_DCM_Reference** field and the associated URL leaf structure are not applicable for the IAV and FAV device classes; they are recommended for the BAV device class.

For more details on the use of the URL, please refer to the DCM Manager chapter of this document.

The **HAVi_DCM_Reference** leaf offset points to a 1212 leaf structure which contains the URL for a remotely stored DCM code unit and its profile. It shall not contain the file name extension of either bytecode unit or profile. It is encoded as follows:

Figure 45. HAVi_DCM_Reference Leaf Structure

leaf length		CRC	
begin URL data			
	end URL data	pad bytes	if necessary

The **HAVi_DCM_Reference** is simply a stream of ASCII text bytes that describe a complete URL. The string shall be NULL-terminated. If the last quadlet is not completely filled with URL data, then null pad bytes shall be added to the end, to fill the last quadlet.

Here's an example of the web address `http://www.mycompany.com/pub/misc/digitalcam` stored in the leaf structure:

Figure 46. Example of HAVi_DCM_Reference Leaf Structure

leaf length		CRC	
"h"	"t"	"t"	"p"
":"	"/"	"/"	"w"
"w"	"w"	"."	"m"
"y"	"c"	"o"	etc.

Note – the format of the leaf containing a URL is not a textual descriptor, and therefore does not have to be formatted as such. The simple embedding of ASCII characters for the URL is sufficient.

9.10.9 HAVi_Device_Icon_Bitmap [HAVi]

The **HAVi_Device_Icon_Bitmap** is a 24-bit offset address which points to a leaf; the leaf contains the bitmap data. The format of the bitmap encoding is described in section 5.12.5.2.

9.11 Examples (Informative)

The following examples illustrate the concepts of defining 1212 ROM structures for HAVi devices.

9.11.1 Using Keys in the Range of 38_{16} to $3F_{16}$

Data fields in the 1212 ROM are segregated by keys, which are indicators of the data which follows. The {key, data} pair allows a controller to parse a 1212 ROM and skip around those fields which it does not understand.

While most of the 6 bit key_IDs are allocated for definition by the IEEE 1212 standard, the meaning of the last eight keys (38_{16}…$3F_{16}$) is defined by the organization or vendor identified by the directory's **Specifier_ID** entry. The meaning of these keys depends on the **Specifier_ID** and the **Version** entry present in the same directory as the keys.

The HAVi keys described in the present specification are defined for exclusive use inside the HAVi Unit Directory. This directory contains as **Specifier_ID** that of the 1394 Trade Association (00 A0 2D) and in the **Version** field the value 01 00 08, identifying the HAVi specification as the one ruling

the use of all the keys in the range of 38_{16} to $3F_{16}$.

For example, HAVi defines key 39_{16} to mean **HAVi_DCM_Key**. Also, Company X might define key 39_{16} to mean private_device_information pointer. When a controller is scanning the ROM, if it sees the HAVi **Specifier_ID** (1394_ta_spec_id), the **Version** (01 00 08 for HAVi) and then key 39_{16}, it understands the meaning as **HAVi_DCM_Key**. If anywhere else in the ROM hierarchy, the controller finds the key 39_{16}, the controller will either understand the key's meaning or it will know that it does not understand the key's meaning and so can ignore the key.

Note that the revised 1212 also specifies extended keys, of 24-bit length, but these are not used in the present version of the HAVi specification.

9.11.2 HAVi 1212 ROM Encoding

The following diagram illustrates a HAVi 1212 ROM, using the fields defined in this document. Note that there may be several other structures required, based on other protocols to which the device conforms. This diagram contains information for a device which also complies to the IEC 61883 protocol.

9.11.2.1 Bus_Info_Block and Root Directory

The ROM header (**Bus_Info_Block**) and root directory appear as follows:

offset **(Base Address FFFF F000 0000_{16})**

Bus_Info_Block

04 00_{16}	04_{16}						crc_length			rom_crc_value				
04 04_{16}	"1394"													
04 08_{16}	irmc	cmc	isc	bmc	pmc	r (3)	cyc_clk_acc (8)		max_rec (4)	r (2)	max_rom (2)	Generation (4)	r	link_spd (3)
04 $0C_{16}$	node_vendor_id (24)											chip_id_hi (8)		
04 10_{16}	chip_id_lo (32)													

Root_Directory

04 14_{16}	root_length	CRC
04 18_{16}	03_{16}	model_vendor_id
04 $1C_{16}$	$0C_{16}$	node_capabilities
	17_{16}	Model_ID
	$D8_{16}$	instance directory offset
	$D1_{16}$	unit directory offset (HAVi or other IEC 61883) (for 61883 compatibility only)
	<< possibly other manufacturer-specific definitions here >>	
	. . . . .	

9.11.2.2 Instance_Directory

The **Instance_Directory** portion of the ROM is referenced by the instance directory offset field in the root directory. It appears as follows.

Figure 47. Instance_Directory (Root Dependent Directory)

directory length	CRC
$D1_{16}$	unit directory offset (HAVi)
$D1_{16}$	unit directory offset (other IEC 61883)
…..	<< possibly other fields >>
…..	…..

9.11.2.3 HAVi_Unit_Directory

The HAVi Unit Directory, referenced by the field labeled “unit directory offset (HAVi)” in the instance directory appears as follows:

Figure 48. HAVi_Unit_Directory (Instance Dependent Directory)

directory length	CRC
12_{16}	Unit_Spec_ID (1394 TA = 00 A0 $2D_{16}$)
13_{16}	Unit_SW_Version (= 01 00 08_{16})
$3D_{16}$	HAVi_Message_Version immediate value (= 00 01 01_{16})
38_{16}	HAVi_Device_Profile value
$9F_{16}$	offset to modifiable descriptor parameter leaf
$B9_{16}$	HAVi_DCM leaf offset
BB_{16}	HAVi_DCM_Profile leaf offset
BA_{16}	HAVi_DCM_Reference leaf offset
BC_{16}	HAVi_Device_Icon_Bitmap leaf offset
…..	<< possibly other fields >>
…..	…….

9.11.2.4 Other IEC_61883_Unit_Directory

Figure 49. Unit_Directory (IEC 61883) Root Dependent Directory

directory length	CRC
12_{16}	Unit_Spec_ID (1394 TA = 00 A0 $2D_{16}$)
13_{16}	Unit_SW_Version (= 01 XX XX_{16})
…..	<< possibly other fields >>
…..	…….

In the **Unit_SW_Version** field the least significant bytes specify any non-HAVi CTS codes specified in [4].

9.11.2.5 Modifiable Descriptor Entries for User Preferred Name

Figure 50. Descriptor Parameter Leaf

leaf length (=2)	CRC
max_name_size (11)	descriptor_address_hi
descriptor_address_lo	

Figure 51. User Preferred Name Leaf in a Modifiable Region of Configuration ROM

leaf length (4)		CRC
descriptor_type(0)	specifier_ID (0)	
width(1)	char_set (1000)	language (0)
0041_{16} "A"		0062_{16} "b"
0043_{16} "C"		0000_{16}

This example shows the name “AbC”, written with 2-byte UNICODE characters.

10 Scenarios

The purpose of this chapter is to describe the interactions between the HAVi system components for various scenarios. The main objective is to help the reader understand the role played by each software component. Explanations of these typical scenario cases are not normative unless indicated otherwise (e.g., by wording including “must” or “shall”).

10.1 IAV or FAV Bootstrap

This scenario describes the operations that occur in system components when a device already connected to the home network is powered-on. It considers the case of an FAV or IAV, because passive devices (BAV or LAV) do not feature HAVi system elements.

10.1.1 System Startup & System Ready

The following rules must be followed during the software bootstrap of a HAVi FAV or IAV device:

- Device Powers ON
- System Elements Start

  Examples:

  - Driver initiated
  - Communication Media Manager (CMM) starts up & is initiated
  - Messaging System/Event Manager/Registry start-up completed and
  - Device Control Module Manager/Stream Manager/Resource Manager, start-up completed (Optional for IAV)
- Messaging System (MS) becomes ready only when all the System Elements are ready.
- Set SDD HAVi_Device_Status Flag to high [DEVICE IS ACTIVE]
- Bus Reset occurs
- Messaging System generates ready `SystemReady` event.

10.1.2 Local Software Elements Register

- Using `Cmm1394::GetGuidList`, the Registry gets the list of GUIDs and extracts the list of IAV and FAV GUIDs (using an implementation dependent mechanism) to obtain the list of other Registry SEIDs on the network.
- Any non-system software element wanting to be known on the network registers through its local Registry (by sending a `Registry::RegisterElement` message).

10.1.3 Interoperation of the Devices within the Network

- DCM Manager enters a DCM installation process according to the protocol described in section 3.6 on DCM Management.

At this stage, the device is ready to interoperate with other devices in the network, since any software element is able to accept incoming messages.

10.2 A BAV or LAV is Plugged into the Network

In this scenario, a new BAV or LAV device is plugged into the 1394 bus (LAV devices not connected to the 1394 network should be handled by FAV/IAV devices in a proprietary manner). The following operations take place on any FAV or IAV device:

- The CMM of each FAV or IAV generates locally a `NetworkReset` event (and also the `NewDevices` event).

- DCM Managers have previously registered interest in the `NetworkReset` event and so receive the event. Using the `Cmm1394::GetGuidList` method, they get the GUIDs of the new and gone devices on the network.

- Each DCM Manager reads the SDD of the newly connected device and detects whether it is a new BAV or LAV.

- One DCM Manager will obtain the right to control this new device, according to the protocol described in section 3.6 on DCM Management (the case of a new guest appearing on the network), and installs a DCM for the new device. The DCM Manager posts a `DcmInstallIndication` event.

- The new DCM components of the new device register through the Registry API after getting software element identifiers from the Messaging System. Then the Registry generates `NewSoftwareElement` events.

- Stream Managers perform their connection restoration process.

10.3 An FAV or IAV is Plugged into the Network

In this scenario, a new FAV or IAV device is plugged into the 1394 bus. Hereafter is described what happens on an FAV or IAV of the network:

- The CMM of each FAV or IAV generates locally a `NetworkReset` event (and also the `NewDevices` event).

- DCM Managers have previously registered interest in the `NetworkReset` event and so receive the event. Using the `Cmm1394::GetGuidList` method, they construct lists of the GUIDs of the new and gone devices on the network.

- The DCM Managers read the SDD of new devices and detects whether they are FAV or IAV.

- According to the protocol described in section 3.6 on DCM Management, the DCM installation process may be re-started since former DCM installations for BAV or LAV devices may have disappeared.

- Stream Managers perform their connection restoration process.

10.4 A BAV or LAV is Removed from the Network

In this scenario, a BAV or LAV device is unplugged from the 1394 bus.

- The CMM of each FAV or IAV generates locally a `NetworkReset` event (and also the `GoneDevices` event).

- DCM Managers have previously registered interest in the `NetworkReset` event and so receive the event. Using the `Cmm1394::GetGuidList` method, they construct lists of the GUIDs of the new and gone devices on the network.

- As explained in section 3.6 on DCM Management, the DCM Manager of the host uninstalls the associated DCM of the BAV or LAV. The DCM components then unregister themselves, through `Registry::UnregisterElement`, and close their message communications via the `MsgClose` API of the Messaging System. (Note that a DCM cannot unregister itself after it has performed a `MsgClose`.) Consequently the Messaging System generates the `MsgLeave` event and the Registry generates the `GoneSoftwareElement` event.

- The DCM Manager posts a `DcmUninstallIndication` event.

- The Messaging System of each device informs locally all software elements that requested notification if the DCM components (with their associated SEIDs) are no longer available.

- Stream Managers perform their connection restoration process.

10.5 An FAV or IAV is Removed from the Network

In this scenario, an FAV or IAV device is unplugged from the 1394 bus:

- The CMM of each FAV or IAV generates locally a `NetworkReset` event (and also the `GoneDevices` event).

- DCM Managers have previously registered interest in the `NetworkReset` event and so receive the event. Using the `Cmm1394::GetGuidList` method, they construct lists of the GUIDs of the new and gone devices on the network.

- The Messaging System of each device informs locally all software elements that requested notification if the DCM components (with their associated SEIDs) hosted by the disappeared device are no longer available.

- Each DCM Manager goes through the DCM allocation process described in section 3.6 (the case of a disappeared host).

- For each DCM previously hosted by the removed device, one DCM Manager will obtain the right to control and install the DCM. The DCM Manager posts a `DcmInstallIndication` event.

- Once installed the DCM registers through the Registry API after getting a software element identifier from the Messaging System. Then the Registry generates a `NewSoftwareElement` event.

- Stream Managers perform their connection restoration process.

10.6 An Application Communicates with an FCM

Once an application has found a functional component corresponding to its needs (by querying the Registry service), this application can contact the FCM to send it commands through the Messaging System. Once the FCM has been contacted, it knows the identifier of the source that sent the command, and thus is able to return responses.

A practical example is given below: an application located on an IAV needs to communicate with a DCM located on another IAV. This DCM can represent the remote IAV itself, or else it can be a BAV/LAV DCM hosted by the IAV.

10.6.1 Initialization

On power-on, the IAV devices perform initialization. The Messaging System is initialized (the `SystemReady` event is generated). The DCMs and FCMs of each device are registered through Registry API (number 1 in Figure 52).

10.6.2 An IAV Searches for a Device or Functional Component

Suppose that the *IAV1* application searches for a VCR. It sends a message to its Registry to query for a VCR FCM (2). The Registry returns a list (which may be empty) of SEIDs of FCMs which fit the criteria (3). Suppose the FCM on *IAV2* is a member of this list. Now the *IAV1* application can contact the *IAV2* FCM and send it commands. The target FCM on reception of a command can send a response according to the definition of the VCR API (section 6.3).

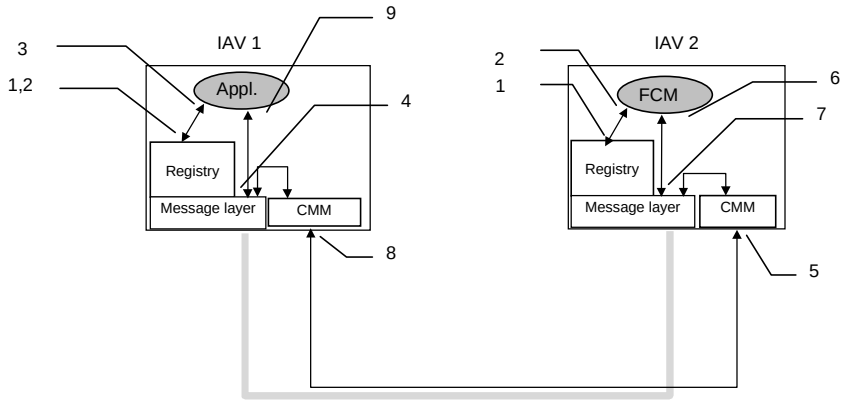

Figure 52. Application and FCM Communication

10.6.3 An IAV Sends an FCM Command

The *IAV1* application uses the Messaging System API to send a `Vcr::Play` command using a SEID previously returned (4). The message layer encapsulates the `Vcr::Play` command and the software element identifier in a message and sends it to the CMM with the GUID of *IAV2* (included within the SEID). The CMM sends the message using IEC 61883/1394 packets with the 1394 network address associated with *IAV2*'s GUID (5).

(In the example shown in Figure 52, the Messaging System makes use of the CMM. This is an implementation choice.)

10.6.4 An IAV Receives an FCM Command

The *IAV2* Messaging System receives the message and checks the destination software element identifier. It then invokes the callback function associated with the destination software element. The *IAV2* FCM receives the command and the software element identifier of the source software element (i.e. application which has sent the message) (6). The FCM then performs the requested action and returns the response as specified in the VCR API (7, 8, 9).

10.7 Two Applications Communicate with the Same DCM

A typical example of this scenario would be two applications offering to two different users access of the same device through the same DCM using the DDI protocol.

- Suppose that neither of the applications is aware of the other. The DCM then has to maintain different (user) contexts in order to fulfill incoming commands.

- The DCM has also to ensure the synchronization that may be needed between these two applications. For example, consider applications A and B controlling a VCR. When user A strikes the PLAY key, the PLAY button must be shown pressed on both graphical UIs (A *and* B) in order to maintain coherence.

10.8 A DCM Communicates with its Target

When a DCM communicates with its target BAV or LAV device, communication is made in a private manner, on the 1394 bus, using CMM services.

This scenario can be illustrated by the example presented below: the DCM of the BAV has been installed within an FAV.

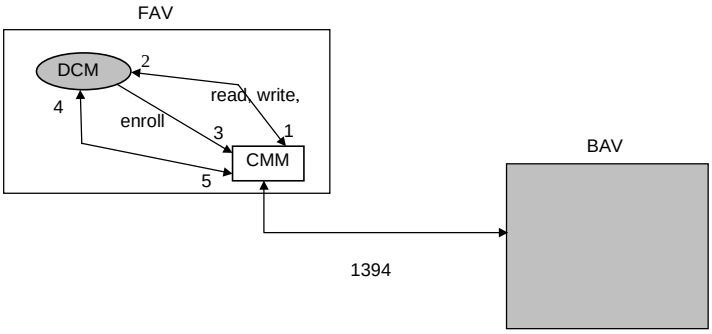

Figure 53. DCM and Target Communication

- When a DCM wants to send an outgoing command to its device, it calls the appropriate function of 1394 CMM API (Read, Write or Lock) (1, 2).
- To receive an incoming command, the DCM uses `Cmm1394::EnrollIndication` (3).
- When an incoming message from the BAV arrives, the CMM will dispatch the message to the DCM, using `<Client>::Cmm1394Indication`, relying on the GUID of the sender (4).
- Then the DCM may then send back a response to the device, again using the appropriate function of the CMM (5).

11 Annexes

11.1 HAVi Protocol Types

All HAVi messages include a 8-bit field called the *protocol type*. The values from 0x00 to 0x7f are reserved for HAVi. The currently defined values of this field are listed in the table below.

HAVi Protocol Type	Value
HAVi_RMI	0x00

11.2 HAVi Registry Attributes

A Registry attribute is associated with an 32-bit `AttributeName`. HAVi reserves the range 0x0 to 0x7fff ffff for system attributes. The following table lists the values in this range currently defined.

Attribute Name	Value
ATT_SE_TYPE	0x0000 0000
ATT_VENDOR_ID	0x0000 0001
ATT_HUID	0x0000 0002
ATT_TARGET_ID	0x0000 0003
ATT_INTERFACE_ID	0x0000 0004
ATT_DEVICE_CLASS	0x0000 0005
ATT_GUI_REQ	0x0000 0006
ATT_MEDIA_FORMAT_ID	0x0000 0007
ATT_DEVICE_MANUF	0x0000 0008
ATT_DEVICE_MODEL	0x0000 0009
ATT_SE_MANUF	0x0000 000a
ATT_SE_VERS	0x0000 000b
ATT_AV_LANG	0x0000 000c
ATT_USER_PREF_NAME	0x0000 000d

11.3 HAVi Software Element Types

The HAVi Registry includes an attribute for storing the "software element type" of each registered software element. This attribute is a 32-bit value and can be used to distinguish between software elements. HAVi specifies the following values of this field. *Note* – for system software elements, the two low order bytes of the software element type are the same as the software handle given in Annex 11.4. Software elements whose software element type is in the range 0x0 to 0x007f ffff are trusted (those with software element type outside this range may or may not be trusted).

HAVi reserves the software element types from 0x0 to 0x7fff ffff. Values above this may be used by application developers and device vendors to create vendor extensions based on the `VendorId`, as stored in the Registry, of the software element.

HAVi Software Element Type	ATT_SE_TYPE Value	Trusted	System Element
MESSAGING_SYSTEM	0x0000 0000	yes	yes
COMMUNICATION_MEDIA_MANAGER	0x0000 0001	yes	yes
REGISTRY	0x0000 0002	yes	yes
EVENT_MANAGER	0x0000 0003	yes	yes
DCM_MANAGER	0x0000 0004	yes	yes
STREAM_MANAGER	0x0000 0005	yes	yes
RESOURCE_MANAGER	0x0000 0006	yes	yes
GENERIC_FCM	0x0000 0100	yes	no
TUNER_FCM	0x0000 0101	yes	no
VCR_FCM	0x0000 0102	yes	no
CLOCK_FCM	0x0000 0103	yes	no
CAMERA_FCM	0x0000 0104	yes	no
AVDISC_FCM	0x0000 0105	yes	no
AMPLIFIER_FCM	0x0000 0106	yes	no
DISPLAY_FCM	0x0000 0107	yes	no
AVDISPLAY_FCM	0x0000 0108	yes	no
MODEM_FCM	0x0000 0109	yes	no
WEBPROXY_FCM	0x0000 010a	yes	no
DCM	0x0000 8000	yes	no
APPLICATION_MODULE	0x0080 0001	possibly	no

11.4 HAVi SEIDs

SEIDs include a 16-bit field called the *software element handle*. The following table gives the values of this field for system software elements (those belonging a system software element type). The values from 0x0000 to 0x00ff are reserved for HAVi system software elements.

HAVi Software Element Type	Software Element Handle
MESSAGING_SYSTEM	0x0000
COMMUNICATION_MEDIA_MANAGER	0x0001
REGISTRY	0x0002
EVENT_MANAGER	0x0003
DCM_MANAGER	0x0004
STREAM_MANAGER	0x0005
RESOURCE_MANAGER	0x0006

11.5 HAVi API Codes

HAVi messages include a 24-bit `OperationCode`. The high order 16 bits are used for an *API code* and the low order 8 bits for an *Operation ID*. The API code is also used in the `Status` structure that is returned by many APIs.

HAVi reserves the API code values from 0x0 to 0x7fff. Values above this may be used by application developers and device vendors to create vendor extensions based on the `VendorId`, as stored in the Registry, of the software element providing the API.

HAVi API Name	API Code
Msg	0x7fff
Version	0x0000
Cmm1394	0x0001
EventManager	0x0002
Registry	0x0003
Dcm	0x0004
Fcm	0x0005
DcmManager	0x0007
StreamManager	0x0008
ResourceManager	0x0009
DdiTarget	0x000a
unused	0x000b
ApplicationModule	0x000c
Tuner	0x000d
Vcr	0x000e
Clock	0x000f
unused	0x0010
Camera	0x0011
AvDisc	0x0012
Amplifier	0x0013
Display	0x0014
Modem	0x0015
WebProxy	0x0016

11.6 HAVi Operation Codes

HAVi messages include a 24-bit `OperationCode`. The high order 16 bits are used for an *API code* and the low order 8 bits for an *Operation ID*. HAVi reserves the Operation ID values from 0x0 to 0x7f when used in combination with a HAVi reserved API code. Values above this may be used by application developers and device vendors to create vendor extensions based on the `VendorId`, as stored in the stored in the Registry, of the software element providing the API.

The following table gives the values of `OperationCode` for messages sent to HAVi software elements.

HAVi Message	API Code	Operation ID
Msg::Ping	0x7fff	0x00
Version::GetVersion	0x0000	0x00
Cmm1394::GetGuidList	0x0001	0x00
Cmm1394::Write	0x0001	0x01
Cmm1394::Read	0x0001	0x02
Cmm1394::Lock	0x0001	0x03
Cmm1394::EnrollIndication	0x0001	0x04
Cmm1394::DropIndication	0x0001	0x05
EventManager::Subscribe	0x0002	0x00
EventManager::Unsubscribe	0x0002	0x01
EventManager::Replace	0x0002	0x02
EventManager::AddEvent	0x0002	0x03
EventManager::RemoveEvent	0x0002	0x04
EventManager::PostEvent	0x0002	0x05
EventManager::ForwardEvent	0x0002	0x06
Registry::RegisterElement	0x0003	0x00
Registry::RetrieveAttributes	0x0003	0x01
Registry::UnregisterElement	0x0003	0x02
Registry::GetElement	0x0003	0x03
Registry::MultipleGetElement	0x0003	0x04
Dcm::GetDeviceIcon	0x0004	0x00
Dcm::GetHuid	0x0004	0x01
Dcm::GetFcmCount	0x0004	0x02
Dcm::GetFcmSeidList	0x00x4	0x03
Dcm::GetDeviceClass	0x0004	0x04
Dcm::GetDeviceManufacturer	0x0004	0x05

Dcm::GetUserPreferredName	0x0004	0x06
Dcm::SetUserPreferredName	0x0004	0x07
Dcm::GetPowerState	0x0004	0x08
Dcm::SetPowerState	0x0004	0x09
Dcm::NativeCommand	0x0004	0x0a
Dcm::GetControlCapability	0x0004	0x0b
Dcm::GetHavletCodeUnitProfile	0x0004	0x0c
Dcm::GetHavletCodeUnit	0x0004	0x0d
Dcm::GetPlugCount	0x0004	0x0e
Dcm::GetPlugStatus	0x0004	0x0f
Dcm::Connect	0x0004	0x10
Dcm::Disconnect	0x0004	0x11
Dcm::GetConnectionList	0x0004	0x12
Dcm::GetChannelUsage	0x0004	0x13
Dcm::GetPlugUsage	0x0004	0x14
Dcm::SetIecBandwidthAllocation	0x0004	0x15
Dcm::IecSprayOut	0x0004	0x16
Dcm::IecTapIn	0x0004	0x17
Dcm::GetSupportedTransmissionFormats	0x0004	0x18
Dcm::GetTransmissionFormat	0x0004	0x19
Dcm::SetTransmissionFormat	0x0004	0x1a
Dcm::GetContentIconList	0x0004	0x1b
Dcm::SelectContent	0x0004	0x1c
Dcm::StopContent	0x0004	0x1d
Dcm::ScheduleReservation	0x0004	0x1e
Dcm::UnscheduleReservation	0x0004	0x1f
Dcm::GetScheduledActionReferences	0x0004	0x20
Dcm::AddVirtualFcm	0x0004	0x21
Dcm::RemoveVirtualFcm	0x0004	0x22
Dcm::GetAvailableStreamTypes	0x0004	0x23
Dcm::GetStreamType	0x0004	0x24
Dcm::SetStreamTypeId	0x0004	0x25
Fcm::GetHuid	0x0005	0x00
Fcm::GetDcmSeid	0x0005	0x01
Fcm::GetFcmType	0x0005	0x02

Fcm::GetPowerState	0x0005	0x03
Fcm::SetPowerState	0x0005	0x04
Fcm::NativeCommand	0x0005	0x05
Fcm::SubscribeNotification	0x0005	0x06
Fcm::UnsubscribeNotification	0x0005	0x07
Fcm::GetPlugCount	0x0005	0x08
Fcm::GetSupportedStreamTypes	0x0005	0x09
Fcm::Wink	0x0005	0x0c
Fcm::Unwink	0x0005	0x0d
Fcm::CanWink	0x0005	0x0e
Fcm::Reserve	0x0005	0x0f
Fcm::Release	0x0005	0x10
Fcm::GetReservationStatus	0x0005	0x11
Fcm::GetWorstCaseStartupTime	0x0005	0x12
Fcm::SetPlugSharing	0x0005	0x13
Fcm::IecAttach	0x0005	0x14
Fcm::IecDetach	0x0005	0x15
DcmManager::SetPreference	0x0007	0x00
DcmManager::GetPreference	0x0007	0x01
DcmManager::GetDeviceIcon	0x0007	0x02
DcmManager::InstallDcm	0x0007	0x03
DcmManager::UninstallDcm	0x0007	0x04
DcmManager::DMInitialization	0x0007	0x05
DcmManager::DMInitialInquiry	0x0007	0x06
DcmManager::DMInquiry	0x0007	0x07
DcmManager::DMCommand	0x0007	0x08
DcmManager::DMGetDcm	0x0007	0x09
StreamManager::FlowTo	0x0008	0x00
StreamManager::SprayOut	0x0008	0x01
StreamManager::TapIn	0x0008	0x02
StreamManager::Drop	0x0008	0x03
StreamManager::GetLocalConnectionMap	0x0008	0x04
StreamManager::GetGlobalConnectionMap	0x0008	0x05
StreamManager::GetConnection	0x0008	0x06
StreamManager::GetStream	0x0008	0x07

ResourceManager::Reserve	0x0009	0x00
ResourceManager::Release	0x0009	0x01
ResourceManager::Negotiate	0x0009	0x02
ResourceManager::ScheduleAction	0x0009	0x03
ResourceManager::UnscheduleAction	0x0009	0x04
ResourceManager::GetLocalScheduledActions	0x0009	0x05
ResourceManager::GetScheduledActionData	0x0009	0x06
ResourceManager::TriggerNotification	0x0009	0x07
ResourceManager::GetScheduledConnections	0x0009	0x08
DdiTarget::Subscribe	0x000a	0x00
DdiTarget::Unsubscribe	0x000a	0x01
DdiTarget::GetDdiElement	0x000a	0x02
DdiTarget::GetDdiPanel	0x000a	0x03
DdiTarget::GetDdiGroup	0x000a	0x04
DdiTarget::GetDdiElementList	0x000a	0x05
DdiTarget::GetDdiContent	0x000a	0x06
DdiTarget::ChangeScope	0x000a	0x07
DdiTarget::UserAction	0x000a	0x08
ApplicationModule::GetIcon	0x000c	0x00
ApplicationModule::GetHuid	0x000c	0x01
ApplicationModule::GetHavletCodeUnitProfile	0x000c	0x02
ApplicationModule::GetHavletCodeUnit	0x000c	0x03
Tuner::GetServiceListInfo	0x000d	0x00
Tuner::GetServiceList	0x000d	0x01
Tuner::SetServiceList	0x000d	0x02
Tuner::GetService	0x000d	0x03
Tuner::GetServiceComponents	0x000d	0x04
Tuner::GetServiceEvents	0x000d	0x05
Tuner::SelectService	0x000d	0x06
Tuner::GetSelectedServices	0x000d	0x07
Tuner::GetCapability	0x000d	0x08
Vcr::Play	0x000e	0x00
Vcr::Record	0x000e	0x01
Vcr::FastForward	0x000e	0x02
Vcr::FastReverse	0x000e	0x03

Vcr::VariableForward	0x000e	0x04
Vcr::VariableReverse	0x000e	0x05
Vcr::Stop	0x000e	0x06
Vcr::RecPause	0x000e	0x07
Vcr::Skip	0x000e	0x08
Vcr::EjectMedia	0x000e	0x09
Vcr::GetState	0x000e	0x0a
Vcr::GetRecordingMode	0x000e	0x0b
Vcr::SetRecordingMode	0x000e	0x0c
Vcr::GetFormat	0x000e	0x0d
Vcr::GetPosition	0x000e	0x0e
Vcr::ClearRTC	0x000e	0x0f
Vcr::GetCapability	0x000e	0x10
Vcr::GetRejectInfo	0x000e	0x11
Clock::GetDateTime	0x000f	0x00
Clock::SetDateTime	0x000f	0x01
Clock::GetTimezone	0x000f	0x02
Clock::SetTimezone	0x000f	0x03
Clock::EnableAutoDST	0x000f	0x04
Clock::IsEnabledAutoDST	0x000f	0x05
Clock::GetCapability	0x000f	0x06
Clock::CreateTimer	0x000f	0x07
Clock::GetTimerState	0x000f	0x08
Clock::SetTimerState	0x000f	0x09
Clock::DeleteTimer	0x000f	0x0a
Camera::Zoom	0x0011	0x00
Camera::Pan	0x0011	0x01
Camera::Tilt	0x0011	0x02
Camera::SetVideoState	0x0011	0x03
Camera::GetVideoState	0x0011	0x04
Camera::Shoot	0x0011	0x05
Camera::GetImageList	0x0011	0x06
Camera::OpenImage	0x0011	0x07
Camera::ReadImage	0x0011	0x08
Camera::CloseImage	0x0011	0x09

Camera::EraseImage	0x0011	0x0a
Camera::GetCapability	0x0011	0x0b
AvDisc::GetItemList	0x0012	0x00
AvDisc::Play	0x0012	0x01
AvDisc::Record	0x0012	0x02
AvDisc::VariableForward	0x0012	0x03
AvDisc::VariableReverse	0x0012	0x04
AvDisc::Stop	0x0012	0x05
AvDisc::RecPause	0x0012	0x06
AvDisc::Skip	0x0012	0x07
AvDisc::InsertMedia	0x0012	0x08
AvDisc::EjectMedia	0x0012	0x09
AvDisc::GetState	0x0012	0x0a
AvDisc::GetFormat	0x0012	0x0b
AvDisc::GetPosition	0x0012	0x0c
AvDisc::Erase	0x0012	0x0d
AvDisc::PutItemList	0x0012	0x0e
AvDisc::GetCapability	0x0012	0x0f
AvDisc::GetRejectInfo	0x0012	0x10
Amplifier::SetVolume	0x0013	0x00
Amplifier::GetVolume	0x0013	0x01
Amplifier::SetMute	0x0013	0x02
Amplifier::GetMute	0x0013	0x03
Amplifier::SetBalance	0x0013	0x04
Amplifier::GetBalance	0x0013	0x05
Amplifier::SetLoudness	0x0013	0x06
Amplifier::GetLoudness	0x0013	0x07
Amplifier::GetCapability	0x0013	0x08
Amplifier::SetEqualizer	0x0013	0x09
Amplifier::GetEqualizer	0x0013	0x0a
Amplifier::GetEqualizerCapability	0x0013	0x0b
Amplifier::SetPresetMode	0x0013	0x0c
Amplifier::GetPresetMode	0x0013	0x0d
Amplifier::GetPresetCapability	0x0013	0x0e
Amplifier::GetAudioLatency	0x0013	0x0f

Display::SetContrast	0x0014	0x00
Display::GetContrast	0x0014	0x01
Display::SetTint	0x0014	0x02
Display::GetTint	0x0014	0x03
Display::SetColor	0x0014	0x04
Display::GetColor	0x0014	0x05
Display::SetBrightness	0x0014	0x06
Display::GetBrightness	0x0014	0x07
Display::SetSharpness	0x0014	0x08
Display::GetSharpness	0x0014	0x09
Display::GetCapability	0x0014	0x0a
Display::GetStandardPictureValue	0x0014	0x0b
Display::SetPresetMode	0x0014	0x0c
Display::GetPresetMode	0x0014	0x0d
Display::GetPresetCapability	0x0014	0x0e
Display::SetScreenMode	0x0014	0x0f
Display::GetScreenMode	0x0014	0x10
Display::SetWindowMode	0x0014	0x11
Display::GetWindowMode	0x0014	0x12
Display::SetActiveWindow	0x0014	0x13
Display::GetActiveWindow	0x0014	0x14
Display::GetWindowRectangle	0x0014	0x15
Display::AssignPlugToDisplay	0x0014	0x16
Display::GetVideoLatency	0x0014	0x17
Modem::AsyncOpen	0x0015	0x00
Modem::IsoOpen	0x0015	0x01
Modem::Send	0x0015	0x02
Modem::Close	0x0015	0x03
Modem::GetCapability	0x0015	0x04
Modem::SetConfiguration	0x0015	0x05
WebProxy::Open	0x0016	0x00
WebProxy::Close	0x0016	0x01
WebProxy::Send	0x0016	0x02
WebProxy::GetCapability	0x0016	0x03

11.7 HAVi Error Codes

The `Status` value returned by HAVi APIs consists of a 16-bit *API code* and a 16-bit *error code*. HAVi reserves the error code values from 0x0 to 0x7fff, the values 0x0 to 0x7f are used for general error messages that may be returned by several APIs. Error codes above 0x7fff may be used by application developers and device vendors to create vendor extensions based on the `VendorId`, as stored in the stored in the Registry, of the software element returning the error.

HAVi Error Name	API Code	Error Code
SUCCESS	any	0x0000
EUNKNOWN_MESSAGE	any	0x0001
EACCESS_VIOLATION	any	0x0002
EUNIDENTIFIED_FAILURE	any	0x0003
ENOT_IMPLEMENTED	any	0x0004
ERESERVED	any	0x0005
EINVALID_PARAMETER	any	0x0006
ERESOURCE_LIMIT	any	0x0007
EPARAMETER_SIZE_LIMIT	any	0x0008
EINCOMPLETE_MESSAGE	any	0x0009
EINCOMPLETE_RESULT	any	0x000a
ELOCAL	any	0x000b
ESTANDBY	any	0x000c
Msg::EFAIL	0x7fff	0x0080
Msg::EALLOC	0x7fff	0x0081
Msg::ESEND	0x7fff	0x0082
Msg::EUNKNOWN	0x7fff	0x0083
Msg::EBUSY	0x7fff	0x0084
Msg::EOVERFLOW	0x7fff	0x0085
Msg::EACK	0x7fff	0x0086
Msg::EELEMENT	0x7fff	0x0087
Msg::ETARGET_REJECT	0x7fff	0x0088
Msg::ESEID	0x7fff	0x0089
Msg::ESOURCE_SEID	0x7fff	0x008a
Msg::EDEST_SEID	0x7fff	0x008b
Msg::ENOT_READY	0x7fff	0x008c
Msg::ESUPER_EXISTS	0x7fff	0x008d
Msg::ETIMEOUT	0x7fff	0x008e

Msg::EPROTOCOL	0x7fff	0x008f
Msg::ESIZE	0x7fff	0x0090
Msg::EDEST_UNREACHABLE	0x7fff	0x0091
Cmm1394::ENOT_READY	0x0001	0x0080
Cmm1394::EADDRESS	0x0001	0x0081
Cmm1394::EHARDWARE	0x0001	0x0082
Cmm1394::ETYPE	0x0001	0x0083
Cmm1394::ESIZE	0x0001	0x0084
Cmm1394::ENOT_INTERESTED	0x0001	0x0085
Cmm1394::ENOT_FOUND	0x0001	0x0086
Cmm1394::EDATA	0x0001	0x0087
Cmm1394::ECONFLICT	0x0001	0x0088
Cmm1394::EUNKNOWN_GUID	0x0001	0x0089
Cmm1394::ETIMEOUT	0x0001	0x008a
Cmm1394::EINVALID_OFFSET	0x0001	0x008b
Cmm1394::EBUSRESET	0x0001	0x008c
Cmm1394::ERETRY	0x0001	0x008d
Cmm1394::EGUID_NOT_EXIST	0x0001	0x008e
EventManager::EEXIST	0x0002	0x0080
EventManager::EDELIVERY	0x0002	0x0081
EventManager::EFORWARDING	0x0002	0x0082
EventManager::ENOT_FOUND	0x0002	0x0083
Registry::ELOCATION	0x0003	0x0080
Registry::EATTRIBUTE_NAME	0x0003	0x0081
Registry::EIDENTIFIER	0x0003	0x0082
Registry::ENETWORK	0x0003	0x0083
Dcm::ENO_PROT	0x0004	0x0080
Dcm::ENO_ADDR	0x0004	0x0081
Dcm::ENOT_SUPPORTED	0x0004	0x0082
Dcm::ENOT_POSS	0x0004	0x0083
Dcm::ENOT_SET	0x0004	0x0084
Dcm::ENO_CONT	0x0004	0x0085
Dcm::ECOMMANDS	0x0004	0x0086
Dcm::ECONNECTIONS	0x0004	0x0087
Dcm::ENONE	0x0004	0x0088

Dcm::ESCHED_OVERLAP	0x0004	0x0089
Dcm::ENOT_RUN	0x0004	0x008a
Dcm::ENO_COMMAND	0x0004	0x008b
Dcm::ESINK_FCM	0x0004	0x008c
Dcm::ESINK_PLUG	0x0004	0x008d
Dcm::ENO_CONNECTION	0x0004	0x008e
Dcm::EINSUFF_BANDWIDTH	0x0004	0x008f
Dcm::EINSUFF_CHANNEL	0x0004	0x0090
Dcm::EDEV_BUSY	0x0004	0x0091
Dcm::ENO_ATTACH	0x0004	0x0092
Fcm::ENO_NOT	0x0005	0x0080
Fcm::ENO_PROT	0x0005	0x0081
Fcm::ENO_ADDR	0x0005	0x0082
Fcm::ENOT_SET	0x0005	0x0083
Fcm::EWAS_WINKING	0x0005	0x0084
Fcm::EWAS_NOT_WINKING	0x0005	0x0085
Fcm::ENOT_SUPPORTED	0x0005	0x0086
Fcm::ENO_RESERVE	0x0005	0x0087
Fcm::ENO_RELEASE	0x0005	0x0088
Fcm::ENO_COMMAND	0x0005	0x0089
Fcm::EATTACH	0x0005	0x008a
Fcm::ENOT_POSS	0x0005	0x008b
DcmManager::EFAIL	0x0007	0x0080
DcmManager::EVOLATILE	0x0007	0x0081
StreamManager::ECONN_ID	0x0008	0x0080
StreamManager::ESOURCE_FCM	0x0008	0x0081
StreamManager::ESINK_FCM	0x0008	0x0082
StreamManager::ESOURCE_PLUG	0x0008	0x0083
StreamManager::ESINK_PLUG	0x0008	0x0084
StreamManager::EUNSUP_TRANSPORT	0x0008	0x0085
StreamManager::EUNSUP_STREAM	0x0008	0x0086
StreamManager::ENO_MATCH_STREAM	0x0008	0x0087
StreamManager::ENO_MATCH_FMT	0x0008	0x0088
StreamManager::ENO_MATCH_TRANSPORT	0x0008	0x0089
StreamManager::ENO_MATCH_DIR	0x0008	0x008a

StreamManager::ESOURCE_BUSY	0x0008	0x008b
StreamManager::ESINK_BUSY	0x0008	0x008c
StreamManager::EDEV_BUSY	0x0008	0x008d
StreamManager::EINSUFF_BANDWIDTH	0x0008	0x008e
StreamManager::EINSUFF_CHANNEL	0x0008	0x008f
StreamManager::EACCESS_VIOLATION	0x0008	0x0090
StreamManager::ENETWORK	0x0008	0x0091
StreamManager::EINVALID_CHANNEL	0x0008	0x0092
StreamManager::ENO_MATCH_BW	0x0008	0x0093
StreamManager::ENO_MATCH_SPEED	0x0008	0x0094
StreamManager::EINVALID_FMT	0x0008	0x0095
StreamManager::ECHANNEL_BUSY	0x0008	0x0096
StreamManager::EASYNC_CHANNEL	0x0008	0x0097
StreamManager::ESTATICBW	0x0008	0x0098
StreamManager::EBROADCAST	0x0008	0x0099
StreamManager::ERESERVED_SOURCE	0x0008	0x009a
StreamManager::ERESERVED_SINK	0x0008	0x009b
StreamManager::EDEV_CONN	0x0008	0x009c
StreamManager::ESHARE	0x0008	0x009d
StreamManager::EANY_CHANNEL	0x0008	0x009e
ResourceManager::ECOMM_CHECK	0x0009	0x0080
ResourceManager::ECONT_SEID	0x0009	0x0081
ResourceManager::ETIME	0x0009	0x0082
ResourceManager::EINV_PLUG	0x0009	0x0083
ResourceManager::EINSUFF_BANDWIDTH	0x0009	0x0084
ResourceManager::EINV_INDEX	0x0009	0x0085
ResourceManager::EMISSING_RES	0x0009	0x0086
ResourceManager::ENOT_SUPPORTED	0x0009	0x0087
ResourceManager::EREJECTED	0x0009	0x0088
ResourceManager::ESCHED_OVERLAP	0x0009	0x0089
ResourceManager::ETRIGG_SEID	0x0009	0x008a
ResourceManager::ERESERVE_FAILED	0x0009	0x008b
DdiTarget::ENO_DEI	0x000a	0x0080
DdiTarget::ENO_SUB	0x000a	0x0081
DdiTarget::ENO_PANEL	0x000a	0x0082

DdiTarget::ENO_GROUP	0x000a	0x0083
DdiTarget::ENOT_CUR	0x000a	0x0084
Tuner::ELIST	0x000d	0x0080
Tuner::ELOCATOR	0x000d	0x0081
Tuner::EUNAVAILABLE	0x000d	0x0082
Tuner::EPERIOD	0x000d	0x0083
Tuner::EPLUG	0x000d	0x0084
Tuner::EREMOVE	0x000d	0x0085
Tuner::ENOT_COMPAT	0x000d	0x0086
Tuner::ELIST_TYPE	0x000d	0x0087
Tuner::ESERVICE	0x000d	0x0088
Vcr::EREJECTED	0x000e	0x0080
Vcr::ENOT_SUPPORTED	0x000e	0x0081
Clock::EUNSET	0x000f	0x0080
Clock::ESET	0x000f	0x0081
Clock::EZONE	0x000f	0x0082
Clock::EAUTO_DST	0x000f	0x0083
Clock::ENO_FREE	0x000f	0x0084
Clock::ETIMER	0x000f	0x0085
Clock::ENOT_OWNER	0x000f	0x0086
Camera::EREJECTED	0x0011	0x0080
AvDisc::EREJECTED	0x0012	0x0080
AvDisc::ENOT_SUPPORTED	0x0012	0x0081
Modem::ENETWORK	0x0015	0x0080
Modem::EBUSY	0x0015	0x0081
Modem::ESETUP	0x0015	0x0082
Modem::EINVALID_MODE	0x0015	0x0083
Modem::ENUM_CONN	0x0015	0x0084
Modem::EFORBIDDEN	0x0015	0x0085
Modem::EINVALID_PLUG	0x0015	0x0086
Modem::ESIZE	0x0015	0x0087
Modem::EFAILED	0x0015	0x0088
Modem::ECID	0x0015	0x0089
Modem::ECONN	0x0015	0x008a
WebProxy::EPROTOCOL	0x0016	0x0080

WebProxy::ESIZE	0x0016	0x0081
WebProxy::EFAILED	0x0016	0x0082
WebProxy::ECID	0x0016	0x0083
WebProxy::ENUM_CONN	0x0016	0x0084
WebProxy::ENETWORK	0x0016	0x0085
WebProxy::EADDRESS	0x0016	0x0086

11.8 HAVi FCM Attribute Indicators

The FCM notification mechanism is based on the use of 16-bit “attribute indicators” which refer to state variables. Each FCM must specify the attribute indicators it supports and the bit syntax of the corresponding attribute value.

HAVi reserves the attribute indicator values from 0x0 to 0x7fff. Values above this may be used by application developers and device vendors to create vendor extensions based on the `VendorId`, as stored in the Registry, of the FCM providing the attribute.

FCM API Name	FCM Attribute	FCM Attribute Indicator	FCM Attribute Value Syntax
Vcr	`currentState`	0x0001	`VcrCurrentState`
Vcr	`recordingMode`	0x0002	`VcrRecordingMode`
Vcr	`counterSet`	0x0003	`boolean`
Vcr	`condensation`	0x0004	`boolean`
Clock	`dateTime`	0x0001	`DateTime`
Clock	`timezone`	0x0002	`Timezone`
Clock	`DSTEnabled`	0x0003	`boolean`
Camera	`videoState`	0x0001	`boolean`
Camera	`zoom`	0x0002	`ZoomOperation`
Camera	`pan`	0x0003	`PanOperation`
Camera	`tilt`	0x0004	`TiltOperation`
AvDisc	`currentState`	0x0001	`AvDiscCurrentState`
Amplifier	`volume`	0x0001	`octet`
Amplifier	`mute`	0x0002	`boolean`
Amplifier	`balance`	0x0003	`octet`
Amplifier	`loudness`	0x0004	`boolean`
Amplifier	`equalizer`	0x0005	`sequence<octet>`
Display	`contrast`	0x0101	`octet`
Display	`tint`	0x0102	`octet`
Display	`color`	0x0103	`octet`
Display	`brightness`	0x0104	`octet`
Display	`sharpness`	0x0105	`octet`
Display	`screenMode`	0x0106	`ScreenMode`
Display	`windowMode`	0x0107	`WindowMode`
Display	`activeWindow`	0x0108	`octet`
Display	`presetMode`	0x0109	`DisplayPresetMode`
Display	`windowRectangle`	0x010a	`WindowRectangle`

Modem	disconnection	0x0001	ModemDisconnection
Modem	callAccept	0x0002	ModemCallAccept
WebProxy	disconnection	0x0001	WebProxyDisconnection

11.9 HAVi System Event Types

An event type is associated with an `EventId`. For system events (`EventId` is a `SystemEventId`) events are identified by a 16-bit *base event number*. The following table lists the base event number for all system event types used by HAVi software elements.

HAVi System Event Type	Posted By	Distribution	Base Event Number
NewDevices	Communication Media Manager	local	0x0001
GoneDevices	Communication Media Manager	local	0x0002
NetworkReset	Communication Media Manager	local	0x0003
SystemReady	Messaging System	global	0x0004
MsgLeave	Messaging System	global	0x0005
MsgTimeout	Messaging System	global	0x0006
MsgError	trusted	global	0x0007
NewSoftwareElement	Registry	global	0x0008
GoneSoftwareElement	Registry	global	0x0009
UserPreferredNameChanged	DCMs	global	0x000a
PowerStateChanged	DCMs, FCMs	global	0x000b
PowerFailureImminent	DCMs, FCMs	global	0x000c
DeviceConnectionAdded	DCMs	global	0x000d
DeviceConnectionDropped	DCMs	global	0x000e
DeviceConnectionChanged	DCMs	global	0x000f
TransmissionFormatChanged	DCMs	global	0x0010
BandwidthRequirementChanged	DCMs	global	0x0011
ContentListChanged	DCMs	global	0x0012
StreamTypeChanged	DCMs	global	0x0013
ReserveIndication	FCMs	global	0x0014
ReleaseIndication	FCMs	global	0x0015
PlugSharingChanged	FCMs	global	0x0016
DcmInstallIndication	DCM Manager	global	0x0017
DcmUninstallIndication	DCM Manager	global	0x0018
ConnectionDropped	Stream Manager	global	0x0019
ConnectionAdded	Stream Manager	global	0x001a
ConnectionChanged	Stream Manager	global	0x001b
InvalidScheduledAction	DCM, Resource Manager	global	0x001c
AbortedScheduledAction	Resource Manager	global	0x001d
ErroneousScheduledAction	Resource Manager	global	0x001e

TunerServiceChanged	Tuner	global	0x001f
VcrStateChanged	VCR	global	0x0020
CameraVideoStateChanged	Camera	global	0x0021
AvDiscItemListChanged	AV Disc	global	0x0022
AvDiscStateChanged	AV Disc	global	0x0023
GuidListReady	Communication Media Manager	local	0x0024

11.10 HAVi Media Formats

A media format is associated with a 64-bit `MediaFormatId`. For media formats specified by HAVi, only the least 40 bits are available, the remaining are set to 0. The following table lists the value of the 40 bits used by HAVi media formats, these are divided into one 8-bit field, *format category*, and two 16-bit fields, *major type* and *minor type*. Format category is a broad classification – examples are tape or disc. A major type identifies a particular family within a format category, the CD family or the MD family for example. A minor type then identifies a particular format within a family, CD-DA and Video CD would be examples for the CD family.

HAVi Media Format	Category	Major Type	Minor Type
NO_MEDIA_PRESENT	0xff	0xffff	0xffff
UNKNOWN_FORMAT	0x00	0x0000	0x0000
TAPE__NO_MEDIA_PRESENT	0x01	0xffff	0xffff
TAPE__UNKNOWN_FORMAT	0x01	0x0000	0x0000
TAPE__DV	0x01	0x0001	0x0001
TAPE__DV__SMALL	0x01	0x0001	0x0002
TAPE__VHS	0x01	0x0002	0x0001
TAPE__VHS__S_VHS	0x01	0x0002	0x0002
TAPE__VHS__D_VHS	0x01	0x0002	0x0003
TAPE__VHSC	0x01	0x0003	0x0001
TAPE__VIDEO8	0x01	0x0004	0x0001
TAPE__HI8	0x01	0x0005	0x0001
DISC__NO_MEDIA_PRESENT	0x02	0xffff	0xffff
DISC__UNKNOWN_FORMAT	0x02	0x0000	0x0000
DISC__CD__DA	0x02	0x0001	0x0001
DISC__CD__VIDEO	0x02	0x0001	0x0002
DISC__CD__PHOTO	0x02	0x0001	0x0003
DISC__CD__ROM	0x02	0x0001	0x0004
DISC__CD__R	0x02	0x0001	0x0005
DISC__MD__AUDIO	0x02	0x0002	0x0001
DISC__MD__PICTURE	0x02	0x0002	0x0002
DISC__DVD	0x02	0x0003	0x0001
DISC__DVD__R	0x02	0x0003	0x0002
DISC__DVD__RAM	0x02	0x0003	0x0003
DISC__DVD__RW	0x02	0x0003	0x0004
DISC__HDD	0x02	0x0004	0x0001

11.11 HAVi Stream Types

A stream type is associated with a `StreamTypeId`, which is a struct containing a `VendorId` field and a `typeNo` field. For stream types specified by HAVi, the `VendorId` field is set to all zeros. The 16 bits of the `typeNo` field are divided into a *major type number* and *minor type number* each of 8 bits. The following table lists the value of the major type and minor type for the HAVi stream types.

HAVi Stream Type	Major Type	Minor Type	Description
UNKNOWN_STREAM	0x00	0x00	
CABLE_STREAM	0x01	0x00	
NO_SIGNAL	0x02	0x00	
DIGITAL_AUDIO__UNKNOWN_STREAM	0x04	0x00	
DIGITAL_AUDIO__CD_PCM	0x04	0x01	
DIGITAL_AUDIO__DVC_NLPCM	0x04	0x02	
DIGITAL_AUDIO__DVD_PCM	0x04	0x03	
DIGITAL_AUDIO__DSS_PCM	0x04	0x04	
DIGITAL_AUDIO__MD_ATRAC	0x04	0x05	
DIGITAL_AUDIO__MPEG1_LAYER_II	0x04	0x06	
DIGITAL_AUDIO__MPEG1_LAYER_III	0x04	0x07	
DIGITAL_AUDIO__MPEG2	0x04	0x08	
DIGITAL_AUDIO__AC3	0x04	0x09	
DIGITAL_AUDIO__DTS	0x04	0x0a	
DIGITAL_VIDEO__UNKNOWN_STREAM	0x05	0x00	
DIGITAL_VIDEO__YUV_422	0x05	0x01	SDI / CCIR 601
DIGITAL_VIDEO__MPEG1	0x05	0x02	
DIGITAL_VIDEO__MPEG2_MP_ML	0x05	0x03	
DIGITAL_VIDEO__MPEG2_MP_HL	0x05	0x04	
DIGITAL_VIDEO__DVC	0x05	0x05	
DATA_STREAM__UNKNOWN_STREAM	0x06	0x00	
DATA_STREAM__DVB	0x06	0x01	
DATA_STREAM__DSS	0x06	0x02	
DATA_STREAM__DVD	0x06	0x03	
DATA_STREAM__ATSC	0x06	0x04	
DATA_STREAM__DAB	0x06	0x05	
MULTIPLEX__UNKNOWN_STREAM	0x07	0x00	
MULTIPLEX__MPEG2_TS	0x07	0x01	MPEG2 transport stream as defined in MPEG-2 Systems (ISO/IEC 13818-1)
MULTIPLEX__MPEG2_PS	0x07	0x02	MPEG2 program stream as defined in MPEG-2 Systems (ISO/IEC 13818-1)
MULTIPLEX__DVB	0x07	0x03	complete DVB multiplex

MULTIPLEX__DVB_PTS	0x07	0x04	partial DVB multiplex as defined in EN 300 468 V1.3.1
MULTIPLEX__DSS	0x07	0x05	
MULTIPLEX__DVD	0x07	0x06	
MULTIPLEX__ATSC	0x07	0x07	complete ATSC multiplex
MULTIPLEX__ATSC_SPTS	0x07	0x08	ATSC single program transport stream as defined in EIA-775
MULTIPLEX__DAB	0x07	0x09	
MULTIPLEX__DVC	0x07	0x0a	

11.12 HAVi CABLE Transmission Formats

CABLE transmission formats are identified by 16-bit values. The following table lists the CABLE transmission formats used by HAVi.

HAVi CABLE Transmission Format	Value
CABLE_FORMAT	0x0000

11.13 HAVi Image Types

An image type is associated with a 40-bit `ImageTypeId`. For image types specified by HAVi, only the 16 least significant bits are available, the remaining are set to 0. The following table lists the value of the 16 bits used for the HAVi image types.

HAVi Image Type	Value
UNKNOWN_IMAGE	0x0000
HAViRAW	0x0001
PNG	0x0002
GIF87	0x0003
GIF89	0x0004
JPEG	0x0005
TIFF	0x0006
DIB	0x0007

11.14 HAVi Transport Types

Transport types have a 16-bit identifier. The following table gives the value of these identifiers for the three transport types used by HAVi.

HAVi Transport Type	Value
CABLE	0x0001
INTERNAL	0x0002
IEC61883	0x0003

11.15 HAVi DDI Element Types

DDI element types have a 16-bit identifier. HAVi reserves the element type values from 0x0 to 0x7fff. DDI element type values above 0x7fff may be used by application developers and device vendors. The following table gives the value of these identifiers for the DDI element types used by HAVi.

DdiElementType	Value
DDI_PANEL	0x0001
DDI_HELP_PANEL	0x0002
DDI_ALERT_PANEL	0x0003
DDI_GROUP	0x0004
DDI_PANELLINK	0x0005
DDI_BUTTON	0x0006
DDI_BASICBUTTON	0x0007
DDI_TOGGLE	0x0008
DDI_ANIMATION	0x0009
DDI_SHOWRANGE	0x000a
DDI_SETRANGE	0x000b
DDI_ENTRY	0x000c
DDI_CHOICE	0x000d
DDI_TEXT	0x000e
DDI_STATUS	0x000f
DDI_ICON	0x0010

11.16 HAVi DDI Optional Attributes

DDI optional attribute types have a 16-bit identifier. HAVi reserves the optional attribute type values from 0x0 to 0x7fff. Optional attribute values above 0x7fff may be used by application developers and device vendors. The following table gives the value of these identifiers for the DDI optional attributes used by HAVi.

OptionalAttribute	Value
POSITION	0x0000
SAFETY_AREA_POSITION	0x0001
BACKGROUND_COLOR	0x0002
BACKGROUND_PATTERN	0x0003
BACKGROUND_PICTURE_LINK	0x0004
AUDIO_VIDEO	0x0005
AUDIO	0x0006
DEVICE_ICON_BITMAP	0x0007
CONTENT_ICON_BITMAP	0x0008
PRESSED_BITMAP_LINK	0x0009
RELEASED_BITMAP_LINK	0x000a
HOTLINK	0x000b
FOCUS_NAVIGATION	0x000c
INITIAL_FOCUS	0x000d
SHOW_WITH_PARENT	0x000e
TITLE	0x000f
FONTSIZE	0x0010
VALUE_OFFSET	0x0011
VALUE_POWER10	0x0012
MAX_LABEL	0x0013
MIN_LABEL	0x0014
CENTER_LABEL	0x0015
UNIT_LABEL	0x0016
FOCUS_SOUND_LINK	0x0017
PRESSED_SOUND_LINK	0x0018
RELEASED_SOUND_LINK	0x0019
SELECT_SOUND_LINK	0x001a
HELP_PANEL_LINK	0x001b
PLAYBACK_DURATION	0x001c
RECORDED_DATETIME	0x001d
BROADCAST_DATETIME	0x001e

11.17 HAVi Comparison Operators

Both the Registry and FCMs support a mechanism for comparing attribute values. Several comparison operators may be used, each identified by a 16-bit quantity as specified below. The most significant bit indicates whether the comparison operator acts on byte rows or sequences.

Comparator	Value	Meaning
ANY	0x0000	any value
EQU	0x0001	equal
NEQU	0x0002	not equal
GT	0x0003	greater than
GE	0x0004	greater than or equal
LT	0x0005	less than
LE	0x0006	less than or equal
BWA	0x0007	bitwise AND
BWO	0x0008	bitwise OR
SEQU	0x8001	equal on sequences
SNEQU	0x8002	not equal on sequences
SGT	0x8003	greater than on sequences
SGE	0x8004	greater than or equal on sequences
SLT	0x8005	less than on sequences
SLE	0x8006	less than or equal on sequences
SBWA	0x8007	bitwise AND on sequences
SBWO	0x8008	bitwise OR on sequences