STANDARDS AND INFORMATION DOCUMENTS

Call for comment on DRAFT AES standard for Audio applications of networks - Open Control Architecture - Part 1: Framework

This document was developed by a writing group of the Audio Engineering Society Standards Committee (AESSC) and has been prepared for comment according to AES policies and procedures. It has been brought to the attention of International Electrotechnical Commission Technical Committee 100. Existing international standards relating to the subject of this document were used and referenced throughout its development.

Address comments by E-mail to standards@aes.org, or by mail to the AESSC Secretariat, Audio Engineering Society, 60 E 42nd St., New York NY 10165. **Only comments so addressed will be considered**. E-mail is preferred. **Comments that suggest changes must include proposed wording.** Comments shall be restricted to this document only. Send comments to other documents separately. Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

This document will be approved by the AES after any adverse comment received within **six weeks** of the publication of this call on http://www.aes.org/standards/comments/, **2012-11-12**, has been resolved. Any person receiving this call first through the *JAES* distribution may inform the Secretariat immediately of an intention to comment within a month of this distribution.

Because this document is a draft and is subject to change, no portion of it shall be quoted in any publication without the written permission of the AES, and all published references to it must include a prominent warning that the draft will be changed and must not be used as a standard.

[This page intentionally blank]

DRAFT

AES standard for Audio applications of networks - Open Control Architecture - Part 1: Framework

Published by
Audio Engineering Society, Inc.

Abstract

AES70 defines a scalable control-protocol architecture for professional media networks. It addresses device control and monitoring only; it does not define standards for streaming media transport. However, the Open Control Architecture (OCA) is intended to cooperate with various media transport architectures.

AES70 is divided into a number of separate parts. This Part 1 describes the models and mechanisms of the Open Control Architecture. These models and mechanisms together form the AES70 Framework. This document should be read together with Part 2, Class structure and Part 3, TCP/IP communications protocol.

Audio Engineering Society Inc., 551 Fifth Avenue, New York, NY 10176, US.

www.aes.org/standards standards@aes.org

Table of Contents

0 Introduction 6
0.1 General 6
0.2 Architectural goals and constraints 6
0.3 Document conventions 8
1 Scope 8
2 Normative references 8
3 Terms, definitions and abbreviations 8
4 Top level design 10
4.1 General 10
4.2 Object orientation 10
4.2.1 Concept 10
4.2.2 Classes 11
4.2.2.1.3 Proprietary classes 11
4.2.3 Instantiation of classes 14
4.3 Messages 14
4.3.1 General 14
4.3.2 Message delivery services 14
5 Device model 14
5.1 Device configurability 14
5.2 Object addressing 15
5.3 Device model elements 16
5.3.1 General 16
5.3.2 Managers, Workers, and Agents 16
5.4 Worker classes 17
5.4.1 Actuators 17
5.4.2 Sensors 17
5.4.3 Blocks 18
5.4.4 Matrices 21
5.5 Agent classes 25
5.5.1 General 25
5.5.2 OcaNetwork and OcaStreamNetwork 25
5.5.3 OcaGrouper 25
5.5.4 OcaRamper 29
5.5.5 OcaNumericObserver and OcaNumericObserverList 29
5.5.6 OcaLibrary 30
5.5.7 OcaMediaClock 31
5.5.8 OcaEventHandler 31
5.6 Manager classes 31
5.7 Standard Object Numbers (ONo) 32
5.8 Object text identification 32
5.9 Constructing objects 32
5.9.1 General 32
5.9.2 Block Factories 32
5.10 Deleting objects 32
6 Events and subscriptions 33
6.1 Subscriptions, events, emitters and notifications 33
6.1.1 General 33

6.1.2 Reliable or fast subscriptions 33
6.1.3 Subscription deletion 33
6.1.4 Subscription event handler 33
6.2 The PropertyChanged event 33
6.3 Use of numeric observers 34
7 Networking 35
7.1 General 35
7.2 Media Transport Connection Management 36
7.3 OCA Adaptations 36
7.3.1 General 36
7.3.2 Kinds of Media Networks 36
7.3.3 Examples 36
7.3.4 The OcaStreamNetwork Class 37
7.3.5 Mode 1: Stream-based Media Connection Management 37
7.3.6 Mode 2: Channel-based media connection management 38
8 Sessions 39
9 Security 39
10 Concurrency control 39
11 Reliability 40
11.1 General 40
11.2 Availability 40
11.2.1 Keep-alive messages 40
11.2.2 Efficient reinitialization 40
11.3 Robustness 40
12 Device reset 41
13 Firmware and software upgrade 41
13.1 General 41
13.2 Update Types 41
13.3 Update Modes 42
13.4 Update mechanisms 42
13.4.1 General 42
13.4.2 Active Updating 42
13.4.3 Passive Updating 43
Annex A (informative) - Actuator example 44
A.1 General 44
A.2 Properties, Methods, and Events 44
Annex B (informative) - Block examples 46
B.1 Simple microphone channel 46
B.2 Two-channel microphone mixer 46
B.3 Mixer using nested blocks 47
Annex C (informative) - Network connection management examples 48
C.1 Stream-based connection examples 48
C.1.1 General 48
C.1.2 Scenario 1: Input 48
C.1.3 Scenario 2: Simple output 49
C.1.4 Scenario 3: Multiple output connectors 49
C.1.5 Scenario 4: Multiple output streams from one connector 50

C.2 Channel-based connection examples 50
C.2.1 Scenario 5: Input 50
C.2.2 Scenario 6: Output 51
Annex D (informative) - Other Media Network Control Standards 52
D.1 AES64 52
D.2 SMPTE ST 2071 - Media Device Control 52
D.3 Architecture for Control Networks (ACN) 52
D.4 Open Sound Control (OSC) 52
D.5 Ember+ 52

Foreword

This foreword is not part of this document, AES70-1-xxxx, *AES standard for audio applications of networks - Open Control Architecture - Part 1: Framework*.

This document is a member of the three-document set that defines AES70, the Open Control Architecture (OCA). AES70-11 defines the architectural framework for AES70. Other parts define the control repertoire and the specific protocols used.

The development project for this standard was originally proposed by the Open Control Architecture Alliance (OCA Alliance) and initiated in October 2012 as project AES-X210 to be developed in task group SC-02-12-L. The OCA Alliance also contributed the task-group working draft and, as a direct result, there are a number of references to "OCA" in the protocol in order to maintain compatibility with implementations already in the field.

The members of the writing group that developed this document in draft are: J. Berryman, H. Hamamatsu, T. Head, S. Jones, M. Lave, N. O'Neill, M. Renz, M. Smaak, G. van Beuningen, S. van Tienen, E. Wetzell.

J. Berryman led the task group.

Richard Foss
Chair, working group SC-02-12
2015-11-12

Note on normative language

In AES standards documents, sentences containing the word "shall" are requirements for compliance with the document. Sentences containing the verb "should" are strong suggestions (recommendations). Sentences giving permission use the verb "may". Sentences expressing a possibility use the verb "can".

DRAFT

AES standard for Audio applications of networks - Open control architecture - Part 1: Framework

0 Introduction

0.1 General

This document describes the models and mechanisms of the AES70 Open Control Architecture (OCA) for the control and monitoring of media networks. These models and mechanisms together form the AES70 Framework.

AES70 is for system control and monitoring only, and may be integrated with any streaming program transport protocol scheme, as long as the underlying communication network is capable of carrying AES70 control and monitoring traffic.

AES70 does not provide a complete device implementation model. AES70 models the control and monitoring functions of a device, not its entire signal path. If a particular device element has no remotely controllable features, then that element need not be represented in the device's AES70 protocol interface.

0.2 Architectural goals and constraints

AES70 is based upon the following features and requirements:

Functionality

AES70 supports the following functions:

1. Discover the AES70 devices that are connected to the network.
2. Define and undefine media stream paths between devices.
3. Control operating and configuration parameters of an AES70 device.
4. Monitor operating and configuration parameters of an AES70 device.
5. For devices with reconfigurable signal processing and/or control capabilities, define and manage configuration parameters.
6. Upgrade software and firmware of controlled devices. Include features for fail-safe upgrades.

Security

AES70 supports the following security measures for control and monitoring data:

1. Entity authentication
2. Prevention of eavesdropping
3. Integrity protection
4. Freshness - *"Freshness" in this context means certainty that replayed messages in a replay attack on a protocol will be detected as such.*

Scalability

AES70 supports networks with up to at least 10,000 application devices. AES70 imposes minimal restriction on the physical distribution of application devices.

Availability

AES70 supports high availability by offering:

1. Device supervision of AES70 devices.
2. Supervision of network connections to AES70 devices.
3. Efficient network re-initialization following errors and configuration changes.

Robustness

AES70 supports robustness by offering:

1. A mechanism for operation confirmation.
2. A mechanism for handling loss of control data.
3. A mechanism for handling device failure of AES70 devices.
4. Recommendations on network robustness mechanisms that network implementers may use.

Safety compliance

AES70 allows implementations of media networks that conform to life-safety emergency standards.

Compatibility

As AES70 evolves, it will maximize compatibility among its different versions. A controller based on one version of AES70 operates with a device based on another version of AES70 in the following manner:

1. For a device based on an older version of AES70, the newer-version controller will function as if it were based on the same version of AES70 as the device.
2. For a device based on a newer version of AES70, the older-version controller will be able to control and monitor all the functions of the device defined in the controller's version of AES70, and will not interfere with functions defined only in the device's version of AES70.

Analyzability

AES70 defines diagnostic functions that allow access to the following information:

1. Version information of all components, hardware and software, of each device
2. Network parameters of a device - for example, MAC address, IP address
3. Device status (including status of devices' network interfaces)
4. Media stream parameters (for each active receive and/or transmit media stream of a device)
5. Communications errors

0.3 Document conventions

In what follows, the phrase "AES70 supports [*x*]", where [*x*] is some function or feature should be interpreted to mean that AES70 defines one or more mechanisms by which a device will be able to implement feature [*x*] in an AES70-compliant manner.

Numerical values are decimal unless otherwise stated.

A Courier typeface is used to identify **`programmatic names`** to distinguish them from regular text.

Where new terminology is first introduced in body text, the term will be set in an italic typeface.

1 Scope

AES70 defines a scalable control-protocol architecture for the control and monitoring of professional media networks. AES70 addresses device control and monitoring only; it does not define standards for transporting streaming media or for describing media content.

This Part 1 describes the models and mechanisms of the AES70 Open Control Architecture. These models and mechanisms together form the AES70 Framework. This document should be read in conjunction with Part 2, Class Structure and Part 3, TCP/IP communications protocol.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

AES70-2 *AES standard for audio applications of Networks - Open Control Architecture - Part 2: Class structure*, Audio Engineering Society, New York, NY., US.

AES70-3 *AES standard for audio applications of Networks - Open Control Architecture - Part 3: Protocol for TCP/IP Networks*, Audio Engineering Society, New York, NY., US.

ISO/IEC 10646-1 *Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and basic multilingual plane*. International Standards Organization (ISO), Geneva, Switzerland

3 Terms, definitions and abbreviations

For the purposes of this document, the following terms and definitions apply.

3.1
AES70-compliant interface
AES70 interface
a network interface compatible with a protocol defined in compliance with this standard.

3.2
Media device
network-connected equipment that originates or accepts media signals and exposes an AES70-compliant interface to controllers on the network to allow control and/or monitoring of its functions.

3.3
Non-media device
network-connected equipment that does not originate or accept media signals, but exposes an AES70-compliant interface to controllers on the network to allow control and/or monitoring of non-media functions.

3.4
AES70 device
Device
a media device or a non-media device. Where the context is clear, the shorter version of the term is used.

3.5
Non-AES70 device
network-connected equipment that does not expose an AES70-compliant interface.

3.6
Device model
the control model for the device as seen by AES70.

3.7
Controller
network-connected software element whose function is to control and/or monitor devices via an AES70-compliant protocol interface. A controller may be hosted in a dedicated computer, or may be a software element running in a device or in some other environment.

3.8
Control
"control" should generally be interpreted to mean "control and monitoring".

3.9
Control protocol
an application protocol whose purpose is the remote control and monitoring of the application functions of devices attached to a network.

3.10
Media stream
a digital data stream that carries a sampled digital audio or video signal over a network or over a point-to-point connection.

3.11
Open Control Protocol
OCP
a network protocol defined in accordance with the AES70 specification

3.12
OCP.1
protocol that implements AES70 for TCP/IP networks. Standardized in *AES70-3*.

3.13
Protocol data unit
PDU
in a layered system, a unit of data that is specified in a protocol of a given layer and contains data relevant to the operation of that layer.

3.14
Object Number
ONo
an arbitrary 32-bit integer used to identify an OCA control object. Object Numbers are unique within each given device.

3.15
Voltage-controlled amplifier
VCA
an analogue electronic amplifier whose gain is controlled by an external direct-current control voltage input.

3.16
Binary Large Object
BLOB
a contiguous block of binary data whose format and, in some cases, length are outside the scope of this standard.

3.17
Availability
the proportion of time a device is in a functioning condition.

3.18
Robustness
the ability of a device to cope with errors during operation despite abnormalities in signal, control input, network operation, or operating environment.

4 Top level design

4.1 General

AES70 (OCA) supports control and monitoring of AES70 devices at the application level. It does not perform audio or video stream transport, but is designed to integrate with various audio and video signal transport schemes.

The AES70 protocol is extensible, to allow the orderly incorporation of new device types and device upgrades, and generally to support upwards-compatible evolution of function in media networks.

AES70 is intended to support practical media networks that are easy to set up and operate. For simple networks of 100 nodes or fewer using recommended switches, the setup procedure should not require technical staff to have advanced networking knowledge. To this end:

1. AES70 networks can operate using industry-standard data network equipment.
2. AES70 devices can coexist harmlessly with non-AES70 devices.
3. AES70 networks may operate in a secure or unsecure mode, as products and applications require.

4.2 Object orientation

4.2.1 Concept

AES70 describes the control interface of a communicating device as a set of *objects*. Each object is a software element that is an instantiation of a specific *class*, and has a state (that is, a set of data elements called *properties*) and a set of procedural actions (called *methods*) defined by that class.

All actions and features of AES70 protocols are defined in terms of classes. The functional scope of a protocol is equivalent to the functional repertoire of the classes it implements. The set of classes fully determines what types of objects may be instantiated in the communicating devices.

A protocol defined in this way is termed an *object-oriented protocol*, and is defined by the union of the following four sets:

1. Class definitions, which define the types of objects that may exist within devices.
2. Naming and addressing rules, which define how the objects and their attributes are identified.
3. Protocol data unit (PDU) formats, which specify the actual formats of transmitted and received data.
4. Protocol data unit (PDU) exchange rules, which define the communication sequences used to effect information exchange (see 4.3).

NOTE This specification is expressed in object-oriented design terms. However, this does not imply that implementations of those protocols must be programmed in an object-oriented style. The choice of object-based or non-object-based implementation is up to the implementer

4.2.2 Classes

4.2.2.1 General

4.2.2.1.1 Class content

Every AES70 class shall consist of the following:

1. A set of elements (properties, methods, and events - see 4.2.2.3)
2. A unique *class identifier*.
3. Exactly one parent class (except for the root class, which has no parent).

NOTE Standard class names (see AES70-2) all begin with "**`Oca`**".

4.2.2.1.2 Inheritance

AES70 classes shall be defined as nodes in a hierarchical tree-like structure which shall start with single, elemental node (the *root class*), and shall be arranged in order by inheritance. Inheritance means that each class is considered to be a more specialized entity (the *child* class) derived from another, less specialized class (the *parent* class). A class shall exhibit (*inherit*) all the features of its parent, except where the definition of that class specifically overrides an inherited feature.

4.2.2.1.3 Proprietary classes

Over the lifetime of this standard, it will often occur that specific products, especially complex ones, will require specialized control classes that are not appropriate for inclusion in the standard class tree. The standard allows for this through four policies:

1. A proprietary class may inherit from any standard class, or from another proprietary class.
2. A proprietary class should be attached to the standard class tree at the most specific level that is appropriate.
3. Proprietary class numbers shall be used for proprietary classes.
4. Proprietary classes shall be defined in accordance with the inheritance rules listed in 4.2.2.4.

4.2.2.1.4 Containment

In this specification, a class will sometimes be described as *containing* another class. This means that an object of the given class incorporates object of the other class.

If the containing class object is deleted, the objects of the classes it contains shall be deleted.

4.2.2.1.5 Collection

In this specification, a class will sometimes be described as *collecting* another class. This means than an object of the given class incorporates *references* to the objects of the other class.

If the collecting class object is deleted, the objects of the classes it collects shall not be deleted.

4.2.2.2 Class identifiers

A class identifier (*class ID*) shall consist of a *lineage key* and a *version number*.

In the following sections, a class ID is represented as $\{n; i_1 \bullet i_2 \bullet i_3 ...\}$, where n is the class version number, and $i_1 \bullet i_2 \bullet i_3 ...$ is the lineage key, as described below.

4.2.2.2.1 Lineage keys

Each class shall be identified by a hierarchical key of the form $i_1 \bullet i_2 \bullet i_3 ...$ where $i_1 \bullet i_2 \bullet i_3$ **...** shall be positive nonzero integer values that uniquely identify a class within its siblings at a particular level of the class tree. i_1, i_2, i_3 and so on are referred to as *class indices.* Class index values may be non-consecutive.

The lineage key of each class shall consist of a set of class indices which identifies the entire lineage of the class, beginning from the root class, extending down through all related child classes, and ending at the class in question. The key may contain as many class indices as needed to describe the layers of the hierarchy.

For example, for a class *X* whose lineage key is **1•2•12•7**, the lineage key shall be interpreted left-to-right as follows:

- **1** designates the root class.
- **1•2** designates a child of the root class.
- **1•2•12** designates a child of the class whose parent is **1•2.**
- **1•2•12•7** designates class X, a child of the class whose parent is **1•2•12**.

4.2.2.2.2 Standard class IDs

Every class ID includes a version number to uniquely identify the revision level of the class. Standard class IDs shall be allocated in *AES70-2*, which will be revised from time to time to include new classes and new versions of existing classes.

4.2.2.2.3 Proprietary class IDs

Proprietary class IDs may be set by the manufacturer of a device. A device's manufacturer shall be identified in a property of its **`OcaDeviceManager`** object. Controllers of devices that include proprietary objects should query the respective **`OcaDeviceManager`** objects to discover the manufacturers of those devices, and behave accordingly.

> NOTE 1 AES70 does not prevent two manufacturers from using the same proprietary class index value for different objects.

If, in a proprietary class ID $\{n; i_1 \bullet i_2 \bullet i_3 ...\}$ a particular index i_k is proprietary, then all indices to the right of it shall be proprietary as well.

> NOTE 2 This restriction is a corollary of the rule that a standard class shall not inherit from a proprietary class.

Any change to a proprietary class definition shall be identified by a higher class-version number.

4.2.2.3 Methods, properties, and events

AES70 classes shall have elements which allow access to their data and operating states.

The elements of AES70 classes shall be as follows:

Properties	A class shall define properties, which are variables that store the class's specific control or monitoring parameters that are accessible from the control network.
Methods	A class shall define methods, which are procedures that controllers may invoke via AES70 protocol commands to retrieve and change property values, to change class operating states, and to perform other actions.
Events	A class may define one or more events, which are callbacks initiated by devices to inform controllers of specific occurrences. To receive events, controllers shall first *subscribe* to them. See 5.7.

These elements shall be used to define protocol exchanges between devices and controllers. The manner in which class elements shall be represented in communications-protocol elements is specific to each AES70 protocol. For one example, see *AES70-3*.

4.2.2.3.1 Element IDs

In the class tree, every method, property, and event shall be assigned not only a name, but also an *element ID* of the form: ***LLtNN***, where

LL shall be the two-digit level of the class tree at which the class is defined.

For example, the global base class **`OcaRoot`** is defined at level **01**. Children of **`OcaRoot`** will be

defined at level **02**. Grandchildren will be defined at level **03**. And so on.

t shall be a type code: **p** for properties, **m** for methods, **e** for events.

NN shall be a sequence number starting at **01** for each type within each class.

Within each class, element ID values shall be unique.

EXAMPLE 1

01p01 is the first property of a class defined at tree level **01**. In this case, **`OcaRoot`** is the only class defined at level **01**, so **01P01** is the first property of **`OcaRoot`**.

EXAMPLE 2

03m02 is the second method of a class defined at tree level **03**. There are several classes defined at level **03** - this ID would apply to the second method of any of these classes.

NOTE 1 Element ID rules are intended to provide a means for uniquely identifying all the native and inherited methods in any given class, and to allow for future expansion of the tree at any level without duplicating identifiers.

NOTE 2 For an example of this approach, see the class **`OcaGain`** in annex A.

NOTE 3 An element ID may be a property ID, a method ID, or an event ID, depending on the type of element it identifies.

4.2.2.3.2 Protocol invariance

For any given class, the following items shall be constant, regardless of which AES70 protocol is used:

1. The set of properties, methods, and events; and
2. The set of element IDs.

4.2.2.3.3 Text format

All text in AES70 properties, method parameters, and event parameters shall be in UTF-8 format (see ISO/IEC 10646-1).

4.2.2.4 Inheritance and updating rules

Inheritance rules ensure that, by creating new classes from existing classes, new features can be added to the protocol in a manner that does not impact the operation of existing products or systems. Inheritance ensures that new child classes will support, at a minimum, the functions and interfaces of their parents.

In AES-AES70, the rules for inheritance are:

1. Any given class except the root class shall inherit from exactly one other class.
2. A class shall implement all the properties, methods, and events of its parent class.
3. A child class may expand its parent's definition by:
    a. Adding new properties, methods, and/or events; and/or
    b. Enhancing the definitions of existing properties, methods, and/or events, In this case, the enhanced definitions shall support all functions defined by the parent class.
4. A child class's inherited methods and events shall retain the respective element IDs of its parent class.
5. A standard class shall not inherit from a proprietary class.

When updating an existing class, the following rules shall apply:

1. The class version number shall be incremented.
2. The updated class shall implement all the properties, methods, and events of the existing class.
3. The updated class may expand the existing class's definition by:
    a. Adding new properties, methods, and/or events; and/or
    b. Enhancing the definitions of existing properties, methods, and/or events, In this case, the enhanced definitions shall support all functions defined by the existing class.
4. An updated class's methods and events shall retain the respective element IDs of its parent class.

4.2.3 Instantiation of classes

As required to create its network control interface, a device shall instantiate classes as objects. Each such object shall be identified by a unique *Object Number (ONo)*.

4.3 Messages

4.3.1 General

Control and monitoring operations shall be implemented by messages in protocol data units (PDUs) that pass between one object and another object which may be in a different device. An AES70 message shall be of one of the following 3 three types:

Command	request devices to return data and/or perform actions.
Acknowledgement	report success or failure of a command, and return requested data, if any.
Notification	report the occurrence of certain events within devices, and provide related data.

With the exception of the device reset message (clause 12), every AES70 command message shall be acknowledged by a corresponding acknowledgement message. Notification messages shall not be acknowledged.

4.3.2 Message delivery services

AES70 recognizes two services for message delivery: *Reliable* and *Fast*.

The Reliable delivery service shall use an assured means of transport that the network offers. The Reliable service shall not be used for multicasting. For example, in TCP/IP networks, the reliable service uses TCP.

The Fast delivery service may use a lower-overhead, lower-reliability means of transport. The Fast service may be used for multicasting, if the network supports it. For example, in TCP/IP networks, the fast service uses UDP.

All AES70 command and acknowledgement messages shall be transmitted using the Reliable delivery service. Notification messages may use either service. A Fast delivery service may be used by the subscription mechanism, described in clause 7, and shall be used by the device reset mechanism, described in clause 12.

NOTE: For some types of networks, Reliable and Fast services may be identical.

5 Device model

5.1 Device configurability

The configurability of an AES70 device may be *fixed*, *pluggable*, *partially-configurable*, or *fully-configurable* as shown below in Table 1.

Fixed and pluggable devices are termed *static* devices, because controllers cannot change their configurations. Partially-configurable and fully-configurable devices are termed *dynamic* devices, because their configurations can be varied while online.

Details of configuration management are in 5.4.3 and *AES70-2*. Creation and deletion of objects are discussed further in 5.7.

Table 1 - Configurability

Configurability	Type	Description
Fixed	static	The device has a permanently assigned object repertoire and signal-flow topology, defined at time of firmware programming.
Pluggable	static	The object repertoire and signal-flow topology of the device may be changed while the device is offline, by plugging and unplugging of hardware modules, adjustment of physical controls, reloading or readjustment of software, or other manual means.
Partially-configurable	dynamic	Controllers may change the signal-flow topology of the device while online.
Fully-configurable	dynamic	A superset of 'partially-configurable', with the addition that controllers may create and delete objects inside the device while online.

5.2 Object addressing

Within an AES70 device, each object (that is, each instantiation of a specific class) shall be identified uniquely by an *object number* (ONo). Depending on device configurability, ONos shall be assigned at different times, as shown in table 2.

NOTE: In specific implementations, developers may use specific object numbering schemes to optimize device performance and/or to ease object management. For example, a developer might choose a device's object number values to correspond with internal table index values, in order to facilitate rapid decoding of incoming commands.

Table 2 - ONo assignment

Configurability	Time of ONo assignment
Fixed	time of manufacture
Pluggable	device setup time
Partially-configurable	time of manufacture
Fully-configurable	time of object creation

In fully-configurable devices, ONo values shall not be re-used when objects are deleted and recreated without an intervening device reset. In the unlikely event that the ONo address space becomes exhausted, ONo values should be re-used beginning with the oldest.

Device implementers may choose ONo values as desired, subject to the following rules:

1 Each Object Number (ONo) shall be a 32-bit integer.

2 Every object shall have a unique ONo.

3 The ONo value of zero shall not be used.

4 ONo values 01 through 4095 shall be reserved for managers (5.6) and other objects that require predefined object numbers. They shall be reserved for AES70 standardization

5 Different ONos may not refer to the same object.

NOTE 1 Unlike some other media control protocols, ONos are not multi-field (*i•j•k*. ...) values based on some hierarchy. Using a simple 32-bit ONo value is a design choice intended to maximize protocol efficiency and to minimize device overheads in resolving object references, handling command responses, and managing errors.

5.3 Device model elements

5.3.1 General

Figure 1 illustrates the AES70 device model. The boxed elements are objects. Details of the classes that define these objects are defined in *AES70-2*. Here we summarize the available classes, giving examples and focusing on those with particular architectural significance.

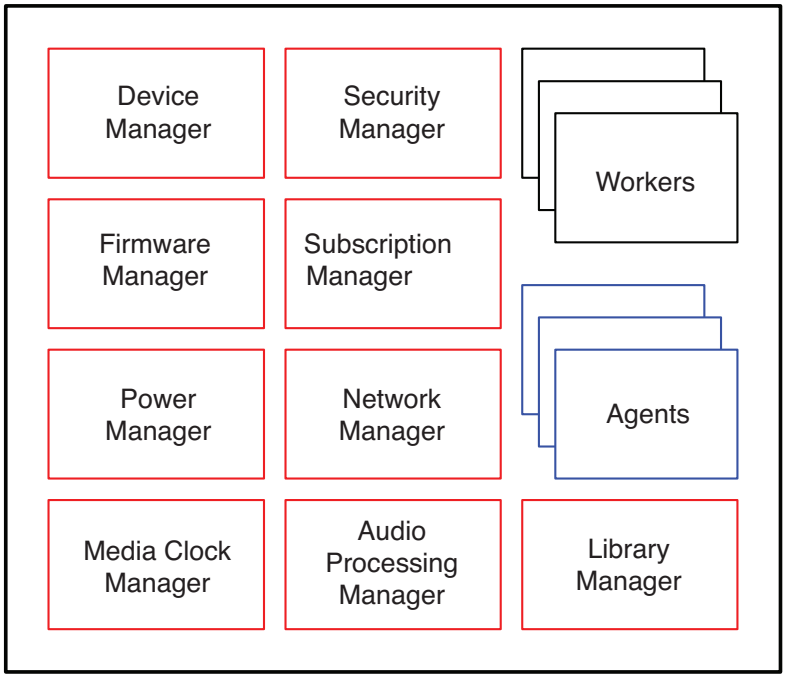


Figure 1 - AES70 device model

5.3.2 Managers, Workers, and Agents

The AES70 device model contains three categories of objects, as follows:

1. **Managers**. Manager objects shall be control objects that affect or report the basic attributes and overall states of the device. Within each device there shall be only one instance of each manager class (that is, one Device Manager, one Security Manager, and so on). Each manager shall have a standard object number.

   Some managers are required, others are optional - see 5.6 and AES70-2.

2. **Workers**. Worker objects shall directly control the application functions of the device - see 5.4. Examples include: audio mute switches, gain controls, equalizers, level sensors, overload sensors; video camera controls, signal properties, image processing parameters, and signal processing functions.

   Workers shall be classified as follows:

Actuators	control application functions (a switch, for example)
Sensors	detect and report signal parameters and other values back to controllers
Blocks and **Block Factories**	allow the assembling of related objects into collections
Matrices	allow collections of objects to be addressed as two-dimensional arrays
Networks	describe digital network connection features

3. **Agents.** Agent objects shall provide indirect control of Workers within a device. An Agent shall not reflect a signal processing function, but instead may affect signal processing parameters in one or more associated Workers, or provide other application control functions. See 5.5.

For example, an Agent named **`OcaGrouper`** implements complex grouping of control parameters in a manner that resembles VCA grouping in analogue systems.

Detailed definitions of all of the classes in these three categories are specified in AES70-2. Rules and concepts governing those definitions are given below.

5.4 Worker classes

5.4.1 Actuators

Actuators shall control signal-processing and housekeeping functions within a device. Actuator classes are detailed in AES70-2. A single complete example is given in annex A.

In any device, any actuator class may be instantiated as many times as required for control of application function.

Examples include:

`OcaGain`	Controls a gain function - see annex A for a more detailed description.
`OcaMute`	Controls a signal mute function.
`OcaSwitch`	Controls a multiposition selector.
`OcaFilterParametric`	Controls a parametric equalizer section.
`OcaDelay`	Controls a signal delay.
`OcaDynamics`	Controls a limiter, compressor, expander, or gate.
`OcaTemperatureActuator`	Controls a temperature setting

5.4.2 Sensors

Sensors shall detect the value of some parameter and transmit it back to controllers. Sensor classes are defined in AES70-2. Examples include:

`OcaLevelSensor`	Senses a signal level
`OcaAudioLevelSensor`	Senses an audio signal level using standard VU or PPM averaging and ballistics
`OcaTemperatureSensor`	Senses a temperature

Sensor values may be transmitted automatically, on a periodic or conditional basis (for example, when it exceeds a defined threshold), by using the **`OcaNumericObserver`** agent (see 5.5.5 and 6.3).

5.4.3 Blocks

5.4.3.1 General

A block shall be a special kind of Worker that may contain Worker objects (including other block objects) and/or Agent objects. A block may also describe a signal-flow topology among Workers it contains. Normative definitions of block classes are in AES70-2.

An object inside a block is referred to as a *member* of that block. The block which contains an object is referred to as the *container* of that object. AES70 blocks may be nested to any depth.

Every Worker and every Agent shall be a member of exactly one block. Every device shall have at least one block, known as the *root block*, that shall contain all of the device's Workers and Agents. In simple devices, all the Workers and Agents may belong to the root block. In more complex devices, Workers and Agents may be deployed into nested blocks.

Manager objects shall not belong to blocks.

Each block shall be described by a class (the block class). The base class for all block classes shall be named **`OcaBlock`**.

> NOTE 1 It is important to remember that blocks are abstract containers which make minimal assumptions about device structure and control flow. There are no predefined block types and no assumptions about what objects blocks might contain. Devices may have any arrangement of blocks or no blocks other than the root block.
>
> NOTE 2 Anticipated uses of blocks include: (1) Providing enumeration functions (5.4.3.2); (2) Storing signal-flow information (5.4.3.6.4); (3) Supporting library functions (5.5.6); (4) For reconfigurable devices, (a) Allowing creation and deletion of objects and signal paths, and (b) providing an organizing mechanism for creation and deletion of aggregate processing functions (5.9).

5.4.3.2 Block enumeration

The set of Workers and Agents that the block contains is termed the block's *object set*. The list of members of an object set is termed an *object list*.

`OcaBlock` shall provide methods for retrieval of the object list of any specified block. Options shall be provided for the device to enumerate directly contained objects only, or to enumerate recursively all directly contained objects plus all objects inside nested blocks to any level.

> NOTE To discover every Worker object in a device, a controller need only request recursive enumeration of the root block.

5.4.3.3 Block management

Fully-configurable devices may allow the creation, modification, and deletion of blocks. AES70 defines methods that controllers may use to perform these functions for such devices.

Deleting a block shall delete all of the objects it contains.

Details of object and block construction are specified in 5.7.

5.4.3.4 Blocks and object addressing

Addressing of objects shall be provided strictly by object number.

> NOTE 1 AES70 does not define a hierarchical object-addressing scheme based on block containment.
>
> NOTE 2 Manufacturers are free to choose object number values that are representative of a device's block structure, if desired. Such choices are outside the scope of AES-AES70.

5.4.3.5 Blocks and control aggregation

Blocks shall not aggregate control functions, such as ganging, grouping, or mastering.

> NOTE AES70 control function aggregation is provided by the **`OcaGrouper`** class. The **`OcaGrouper`** mechanism is independent of block boundaries, and fully supports multiple overlapping grouping functions - see 5.5.3

5.4.3.6 Signal flow

5.4.3.6.1 Ports

For the purpose of defining signal flow, every Worker object may contain one or more *ports.* A port is a data element that describes one input or output signal channel of the processing function that the Worker represents.

An *input port* shall represent a signal flow into the processing function; an *output port* shall represent a signal flow out of the processing function.

> NOTE For readability in what follows, the text will generally describe signal connections between signal processing functions in terms of the objects which represent those functions. For example, to denote a signal connection from the processing function described by object A to the processing function described by object B, the text will read, "a signal connection from object A to object B".

5.4.3.6.2 Block ports

Blocks are Workers, and may therefore have ports. Such ports are termed *block ports*. Block ports shall be intermediate ports that exist to define signal connection points between objects inside the block and objects outside the block. See figure 2, below.

A *block input port* shall be a connection point for signals entering the block. To objects inside the block, a block input port shall appear as a signal source. A *block output port* shall be a connection point for signals leaving the block. To objects inside the block, an output block port shall appear as a signal sink.

An *internal signal path* is a signal path between two objects in the same block. An *external* signal path is a signal path between two objects in different blocks. This is illustrated in figure 2.

5.4.3.6.3 Signal paths

The term *signal path* shall denote a connection between a specific output port and a specific input port in the same device. A signal path's input and output ports may be in different objects or in the same object. A signal path between objects in separate blocks may or may not include block ports.

> NOTE The use of block ports is encouraged, but is not required. AES70 allows direct connections between objects in separate blocks.

5.4.3.6.4 Signal flow of a block

A block's *signal flow* shall comprise the set of all signal paths with at least one end inside the block. The list of signal paths in a signal flow is termed a *signal-flow list*.

The block's signal flow shall exclude signal paths that extend from block ports to other ports outside the block. Such paths belong to the overall containing block. In simple cases, the containing block will be the root block.

Media transport connections between one device and another shall not be considered part of device signal flow. Media connections between devices are described in 7.2.

`OcaBlock` shall provide methods for retrieval of the signal-flow list of any specified block. Options shall be provided to enumerate directly-contained signal paths only, or to enumerate recursively all directly contained signal paths plus all signal paths inside nested blocks to any level.

Figure 2 shows a typical signal flow.

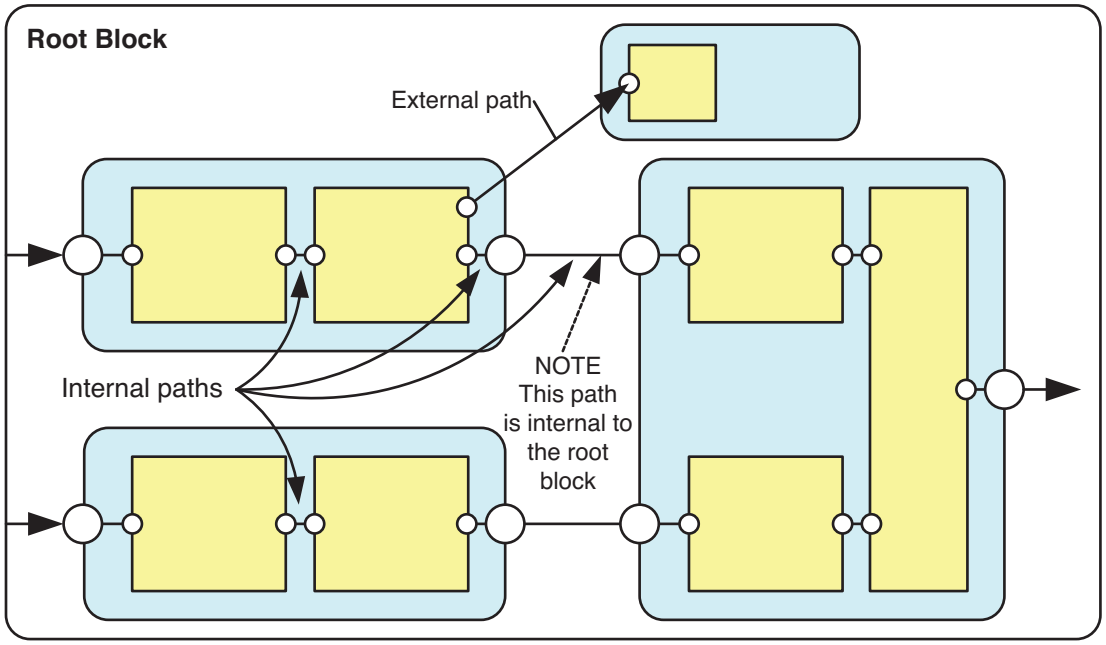


Figure 2 - Signal flow

NOTE The illustrations in this document use small circles for Worker ports, large circles for block ports, and simple lines for signal paths. Blocks are shown with rounded corners, other objects with square corners.

5.4.3.7 Examples

Further examples of blocks and signal flows are given in annex B.

5.4.3.8 Block factories

In fully-configurable devices, controllers shall be able to build blocks, populate them with objects, and connect signal paths among those objects. The primary mechanism for doing this is the *Block Factor*y.

A block factory shall be a Worker whose job is to construct fully populated and "wired" blocks. Each block factory shall be an instance of class **`OcaBlockFactory`** that has been configured to construct one specific kind of block, with a predefined set of objects and signal paths.

Note: Block factories are useful when controllers must create multiple blocks, all of the same kind.

A block factory shall be a container of prototype objects and prototype signal flows which are realized each time the factory constructs a block.

AES70 provides two ways of creating block factories. Either or both of these methods may be implemented, depending on device type.

1. One or more block factories may be defined at time of manufacture or, for pluggable devices, at time of device setup. In this case, these block factories shall provide controllers with a predefined repertoire of block types which they may instantiate.

2. The controller may create block factories, using specific AES70 commands to define the configurations of blocks they will create. In this case, controllers shall be free to define new block types which may be subsequently instantiated.

5.4.3.9 Block creation without block factory

A controller may create a block without using a block factory by instantiating an object of class **`OcaBlock`**, then sending commands to the device to create and interconnect specific objects inside the new block. The **`OcaBlock`** class shall provide the necessary methods for so doing.

5.4.4 Matrices

5.4.4.1 General

The AES70 matrix shall be a generalization of the familiar audio crosspoint matrix concept. AES70 matrices are rectangular arrays of identical Workers that share one or more common input and output busses for the rows and columns, respectively.

5.4.4.2 Matrix addressing

Matrix elements shall be addressed by coordinates. AES70 uses (*x*,*y*) coordinates, where *x* is the horizontal (column) number, and *y* is the vertical (row) number (see figure 3).

Coordinate values shall range from 1 to the number of rows or columns.

The special value zero shall be used to denote the entire row or column. So, for example, (0,2) indicates the entire second row. The use of this feature will be shown in 5.4.4.5.

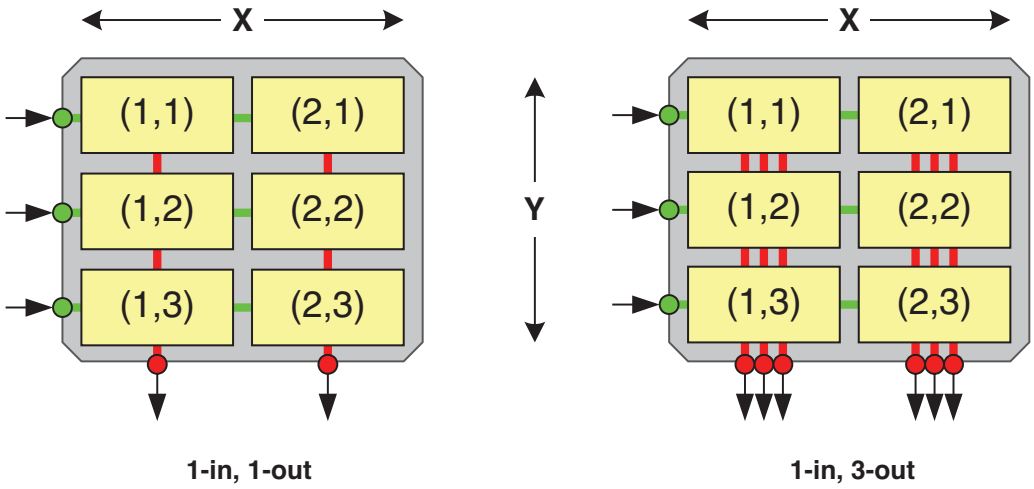


Figure 3 - AES70 matrices

5.4.4.3 Complex matrices

If the Workers in a matrix are switches, the matrix could represent a conventional switching matrix. If the Workers are gain controls, the matrix could represent a conventional mixing matrix. However, in AES70 a matrix's Workers can be of any class, even blocks.

For example, figure 5 shows a matrix of blocks, where each block is as shown in figure 4.

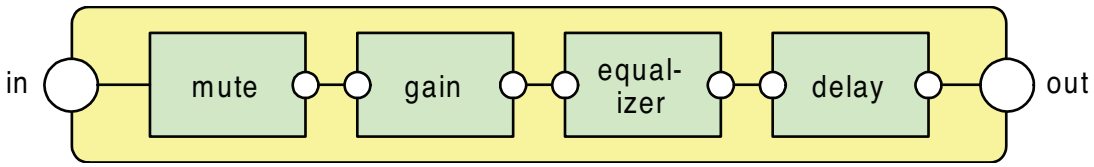


Figure 4 - A complex matrix element

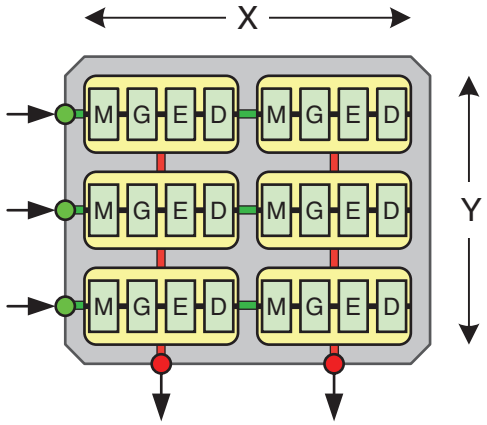


Figure 5 - A matrix of complex elements

5.4.4.4 Matrix structure

A matrix shall be an instance of the **`OcaMatrix`** class, but that instance shall not contain or include the Workers which constitute the matrix elements. The matrix elements shall be instantiated separately, and identified as *members* of the matrix. Their class is termed the *member class* of that matrix. In other words, an AES70 matrix shall collect its members, not contain them.

All of the members of a matrix shall be of the same class, and shall be instantiated in the same device as the matrix instance.

In addition to the matrix and its members, the matrix shall include one additional instance of the member class, termed the *matrix proxy*. The matrix proxy shall be used by controllers to access the settings of matrix members via coordinates.

A complete matrix is illustrated in figure 6, below.

5.4.4.5 Accessing matrix elements

Controllers shall use the **`SetCurrentXY(x,y)`** and **`SetCurrentXYLock(x,y)`** methods to access matrix members using (*x,y*) coordinates:

1. The controller should call the **`OcaMatrix`** method **`SetCurrentXY(x,y)`** or **`SetCurrentXYLock(x,y)`**, specifying a target coordinate set. If **`SetCurrentXYLock(x,y)`** is called, **`OcaMatrix`** shall lock the addressed matrix elements.
2. The controller should call one or more matrix proxy Get or Set methods for the desired property or properties. If Get is called, the current value of the property shall be reported. If Set is called, the new value of the property shall be specified.

If **`SetCurrentXYLock(x,y)`**, was called in step 1, the controller shall call the **`OcaMatrix`** method **`Unlock(x,y)`** after calling to unlock the addressed matrix elements.

The special values x=0 and y=0 may be used to specify aggregate set operations as follows:

`SetCurrentXY(x,y)`	New value will be set in all objects in column x of row *y*
`SetCurrentXY(0,y)`	New value will be set in all objects of row *y*
`SetCurrentXY(x,0)`	New value will be set in all objects of column *x*
`SetCurrentXY(0,0)`	New value will be set in all objects of the whole matrix

Matrix Get methods may be used with x = 0 to retrieve a whole row, or with y = 0 to retrieve a whole column.

Matrix Get methods shall not be used where x=0 AND y=0.

Alternatively, matrix members may be accessed directly using their object numbers. When this occurs, the matrix mechanism is bypassed but shall not be damaged.

AES70 supports the use and management of maximum and minimum values for all object properties. If an aggregate set operation is attempted which would result in the value of any member property to exceed its allowable range, that operation shall be rejected with an error indication. In this case, no member property is changed.

NOTE The maximum and minimum values of properties are specified at object creation time, which may be time of manufacture, time of device hardware setup, or time of operation, depending on the configurability (see 5.1) of the device.

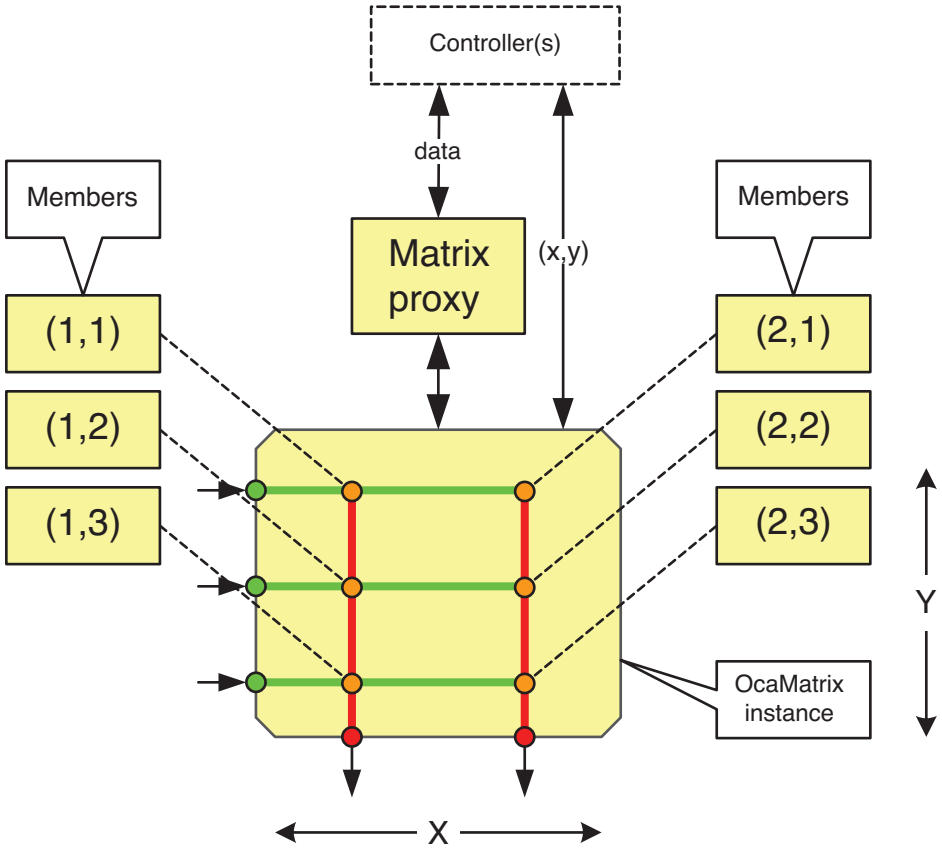


Figure 6 - AES70 matrix structure

5.4.4.6 Matrix signal flow

A matrix shall have input and output signal ports (*matrix ports*) that are separate from the input and output signal ports of its members. Matrix rows shall correspond to matrix input ports; matrix columns shall correspond to matrix output ports. A matrix may have one or more ports per row, and may have one or more ports per column.

In any given matrix, the number of matrix input ports per row, N_{in} shall be the same for all rows, and the number of matrix output ports per column N_{out} shall be the same for all columns.

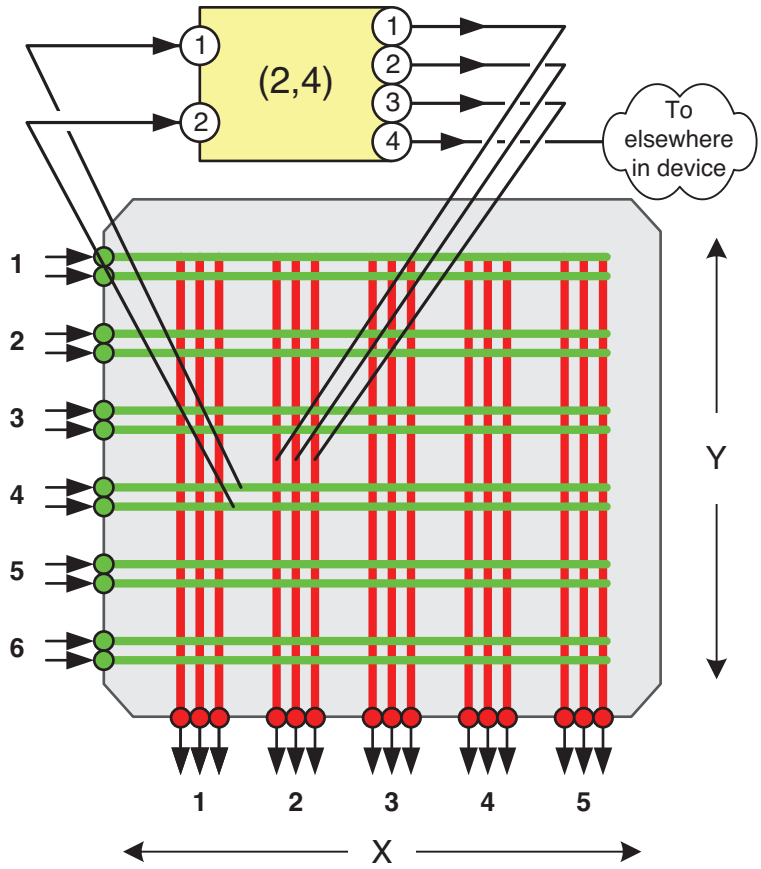
The input and output ports of a member of a matrix should be connected to matrix row and column ports sequentially, according to the following rules:

1. The i^{th} input port of a member in row y should be connected to matrix port $i + N_{in} (y-1)$.
2. The j^{th} output port of a member in column x should be connected to matrix port $j + N_{out} (x-1)$.

A matrix's members should have sufficient numbers of ports to support the above rules. Specifically, each member should have at least N_{in} input ports and N_{out} output ports.

A member may have additional ports that do not connect to matrix ports. The number of such additional ports may vary from member to member.

See figure 7 for an illustration of matrix port relationships.



Ports per row = 2
ports per column = 3

Figure 7 - Matrix port relationships

5.4.4.7 Deployment

AES70 matrices and their proxies shall be instantiated in the same device as their members. For a control aggregation function capable of spanning devices, see **`OcaGrouper`**, clause 5.5.3.

5.4.4.8 Application notes

5.4.4.8.1 Non-mixing applications

Matrices are defined rather generally and may find use not only for the representation of traditional audio matrix mixing, but for other applications which require addressing collections of objects as one- or two-dimensional arrays. Such applications may or may not make use of matrix input and matrix output ports.

For example, a loudspeaker crossover might be represented as a matrix of crossover channels, with each channel being a block containing the usual filters, gain elements, delays, and dynamics controls. In this case, the channels might share a common input which could be represented by a matrix row port. However, they would use separate output ports, so no matrix output ports would be required.

5.4.4.8.2 Non-summing output aggregation

When multiple member outputs connect to a single matrix column port, member signals should be summed to create the final column signal output. However, the AES70 control abstraction does not require such behavior; the matrix object may be used to control other output aggregation algorithms.

5.5 Agent classes

5.5.1 General

This document defines Agent class concepts and semantics. Specific Agent class properties, methods, and events are specified in AES70-2.

5.5.2 OcaNetwork and OcaStreamNetwork

For details of these Agent classes, see clause 7.

5.5.3 OcaGrouper

5.5.3.1 General

`OcaGrouper` shall define an object (*grouper*) that associates property values in such a way as to make them controllable as single values.

> NOTE 1 Groupers support audio control functions variously known as ganging, linking, mastering, submastering, and VCA mastering.

No signals shall pass through groupers - they shall affect control parameters only.

`OcaGrouper` is the root class from which specific grouper classes shall be defined. In what follows, the term *grouper* means an instance of **`OcaGrouper`** or an instance of a child class of **`OcaGrouper`**.

An *actuator grouper* shall provide control of many actuator objects from a single input value, in the manner described below.

A *sensor grouper* shall allow many sensor objects to be observed using a single output value. Sensor groupers will be defined in a subsequent version of this standard.

> NOTE 2 This version of AES70 defines only actuator groupers.

5.5.3.2 Groups

A grouper shall contain one or more *groups.* A group shall be a collection of Workers or Agents, all of the same class. All groups in a grouper shall collect objects of the same class.

Groups shall aggregate whole objects, not individual properties. This aggregation shall maintain the distinctions between the different properties of those objects.

The term *group setpoint* refers to a group's setting for a particular property.

For example, the **`OcaFilterParametric`** class defined in *AES70-2* represents a parametric equalizer with three primary properties - frequency, Q, and passband gain. In a group of **`OcaFilterParametric`** objects, all of these properties shall be grouped separately. Thus, the group shall maintain separate group setpoints for each of frequency, Q, and passband gain.

Group setpoints shall be implemented via the group proxy mechanism described in 5.5.3.4.

5.5.3.3 Overlapping group membership

In a working media system, objects may be members of multiple groups.

For example, in a stereo sound reinforcement system with multi-way loudspeakers, the gain-control object of the left-channel woofer power amplifier could be controlled by: a master gain group, a left-side gain group, and a woofer gain group.

This document refers to groups that share one or more objects as *overlapping groups*.

When a property belongs to an object which is a member of two or more groups, that property's setpoint will depend on the cumulative effect of the corresponding group setpoints of all the groups of which the object is a member.

> NOTE 1 Because all properties have range limits, it is possible that the cumulative effect of overlapping group setpoints might drive a property value out of range. This effect must be prevented or at least managed. In order to manage range limits, a grouper must know about all overlapping groups and which objects belong to which groups, and must have mechanisms which use this knowledge to handle range limit issues in the way the application requires. This is the primary reason why **`OcaGrouper`** has been given the capability of containing multiple groups.
>
> NOTE 2 In order for a grouper's range limit management mechanisms to work in a particular media network, that network should be configured so that all overlapping groups share a common grouper. This is an application guideline.

5.5.3.4 Group structure

5.5.3.4.1 General

A grouper shall be capable of managing range limit issues for both non-overlapping and overlapping groups..

We use the following terminology:

Citizen	An object of which a grouper is aware.
Group	A group that a grouper defines and operates.
Enrollment	The binding of a citizen to a group.
Member	A citizen that is enrolled in a group.
Group proxy	An object that controllers shall use to access group setpoints

A grouper shall provide the following functions:

1. Create or delete groups and group proxies.
2. Register and de-register citizens in the grouper itself.
3. Enroll or de-enroll citizens in groups.
4. Compute and set new property values when group setpoint values change.
5. Manage over-range and under-range group setpoint values in a manner that prevents citizens from being requested to set their properties to out-of-range values.
6. Handle error conditions that arise when connections between grouper and citizen(s) are lost.

All of the citizens of a grouper shall be instances of the same class. A grouper shall collect references to its citizens, but shall not create, destroy, or contain them. A grouper's citizens may reside anywhere on the network - they do not need to be instantiated in the same device as the grouper.

> NOTE It will be helpful to visualize a grouper as a type of data matrix whose rows are groups and columns are citizens, and where each intersection contains information relating to the membership of the citizen (column) in the group (row).

A grouper shall have one or two operating modes - *master-slave mode*, and/or *peer-to-peer mode.* A grouper shall support either one of these modes, or may support both of them. A grouper's mode is determined by the value of its *Mode* property.

5.5.3.4.2 Master-slave mode

In master-slave mode, the parameter values in each group shall be accessed via the group proxy. To change a group setpoint, a controller shall change the corresponding property value of the group proxy.

There shall be exactly one group proxy per group. The class of the group proxy shall be identical to the class of its group's members. As groups are created and deleted, corresponding group proxies shall be automatically instantiated and deleted by the grouper.

Group proxies shall reside in the same device as the grouper.

5.5.3.4.3 Peer to peer mode

In peer-to-peer mode, group proxies shall not be created or used. Instead, the group setpoint shall be changed whenever any member's setpoint is changed. In essence, all the group's members shall behave as though they were group proxies.

5.5.3.4.4 Brief example

Figure 8 shows a typical grouper configuration for controlling a stereophonic set of three-way loudspeaker systems.

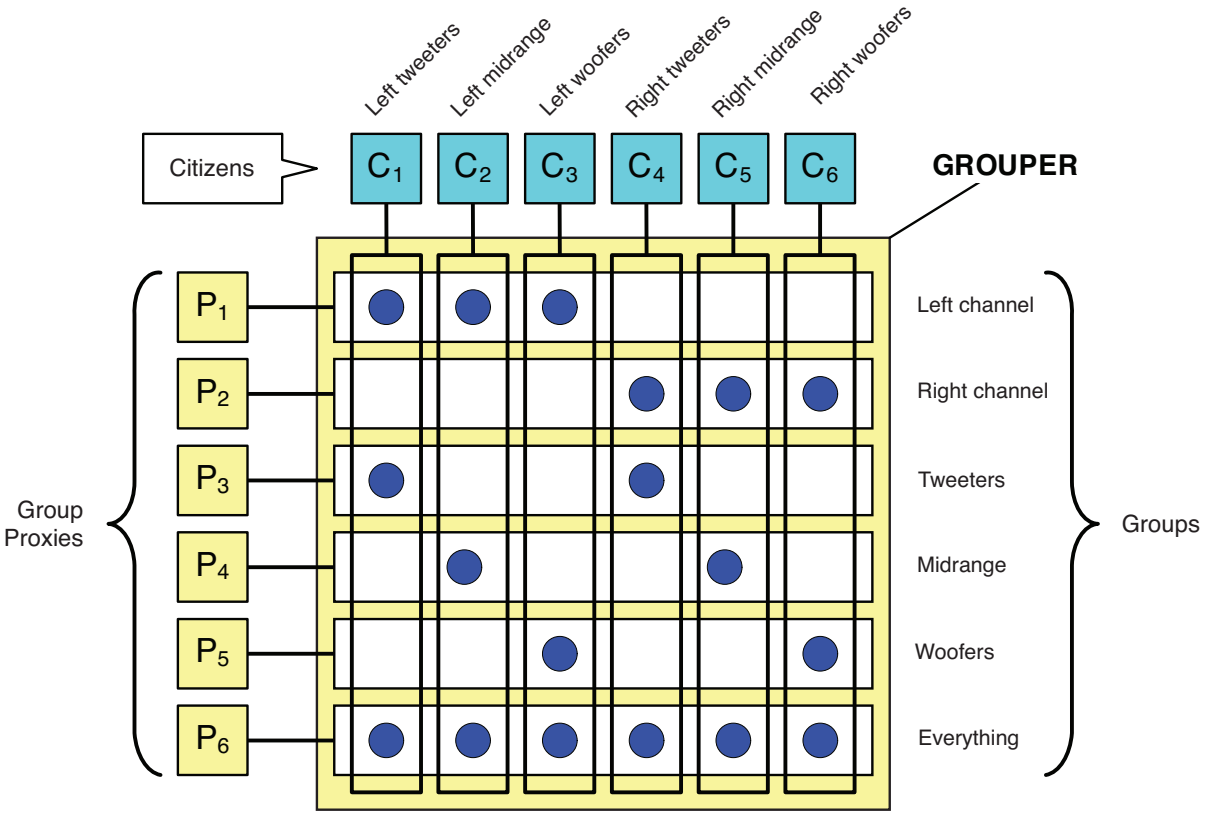


Figure 8 - Typical grouper

5.5.3.5 Aggregation and saturation rules

5.5.3.5.1 General

Citizen setpoints shall be determined by *aggregation rules*, which determine the algorithms by which the setpoint values are computed. Groups shall be governed by *saturation rules*, which determine how over-range conditions are handled. These rules can be expected to vary from citizen class to citizen class, property type to property type, and product to product.

5.5.3.5.2 Aggregation rule options

Groupers may be implemented with a wide range of algorithms to calculate setpoints for individual citizens. Appropriate aggregation rules shall depend on the datatypes of the properties being grouped, on the devices involved, and on the application.

The SUM rule may be used for continuous-value parameters such as gain and delay:

citizen setpoint = sum(setpoints of applicable groups) + offset

where an "applicable group" is one to which the citizen belongs. Offset is a value private for each citizen that makes its setpoint different from the group's setpoint. It may be thought of as a citizen-specific "trim" setting.

In master-slave mode, offset values result from direct controller calls to individual citizen **`Set(...)`** methods, bypassing the grouper. In peer-to-peer mode, this behavior shall not occur.

The LINK rule may be used for ganging discrete-value parameters such as selector switch settings:

citizen setpoint = most recently changed applicable group setpoint

For parameters such as mutes, boolean rules such as AND, OR, and XOR may be useful besides the default LINK rule, for example:

citizen setpoint = XOR(setpoints of applicable groups)

When objects with multiple properties are grouped (for example, parametric equalizers), the aggregation rules may differ among the properties.

5.5.3.5.3 Saturation rule options

A group shall use one of two basic saturation rules:

Saturating — The grouper shall allow all setpoint changes, but shall truncate values sent to individual citizens so that out-of-range conditions are avoided. When group setpoints return to more midrange values, normal operation of the aggregation rule shall resume.

Nonsaturating — The grouper shall refuse to make any setpoint changes that would force any citizen's property out of range.

5.5.3.5.4 Connection loss

When a grouper loses contact with one or more of its citizens, special processing may be needed to preserve system integrity. This is especially important if lost citizens reconnect with the group after a time interval during which group setpoint changes have been made.

`OcaGrouper` shall define an event named **`StatusChange`** to which controllers may subscribe in order to be informed of citizen connection losses.

5.5.3.5.5 Standard OcaGrouper Rules

The standard **`OcaGrouper`** class shall define default saturation, aggregation, and connection-loss rules.

1. Default aggregation rules shall depend on the base datatype of the grouped property. The base datatype is equivalent to its underlying machine datatype. Defaults are shown in table 3.The default saturation rule shall be Nonsaturating.

2. Default connection-loss behavior shall be as follows: After a connection is lost, the standard **OcaGrouper** object shall continue to control any citizens that remain connected; when the connection is restored, the grouper shall update the property values of the lost citizen(s). This update will restore consistency as long as the lost citizen's setpoint values did not change during the time when it was unavailable.

Table 3 - Default aggregation rules

Base Datatype	Aggregation Rule
Float	SUM
Integer	LINK
Bool	LINK
Enum	LINK
String	LINK
Others	(not grouped)

NOTE 1 It is expected that various subclasses of **OcaGrouper** will be implemented to fulfill the range of application requirements. In particular, it is anticipated that different implementations will require differing connection-loss behaviors.

NOTE 2 Groupers may be instantiated in ordinary devices, or in dedicated AES70 control servers, or in system controllers. Where a grouper is deployed is a product or system design choice.

5.5.4 OcaRamper

OcaRamper shall define a mechanism (*ramper*) by which controllers may cause devices to execute incremental or prescheduled parameter changes automatically.

Each ramper shall be bound to a particular Worker property. Ramper parameters shall include target property value, ramp duration, ramp start time, and ramping interpolation law.

Each ramper shall maintain a state property whose value can be monitored by controllers to determine ramping status, for example: waiting to start, in process, complete, or aborted.

NOTE The rationale for **OcaRamper** is as follows: For network performance reasons, it is usually impractical to implement incremental changes (fade-ins, fade-outs, and crossfades) by sending sequences of commands over the network, because the amount of control traffic required would be excessive. Furthermore, timing accuracy of ramping actions will be more precise if not subject to network delays. Also, it may not be possible to implement prescheduled parameter changes in controllers, because application systems may need to execute such changes when controllers are not running.

5.5.5 OcaNumericObserver and OcaNumericObserverList

OcaNumericObserver shall define a *watcher* object that monitors the value of a specified numeric property in another object and, under certain conditions, notifies controllers of the value. **OcaNumericObserver** is part of the Event and Subscription mechanism, and is described in 6.3.

OcaNumericObserverList shall define a watcher object similar to that defined by **OcaNumericObserver**, except that **OcaNumericObserverList** shall be capable of monitoring the values of a given list of numeric properties in other objects.

5.5.6 OcaLibrary

5.5.6.1 Parameter sets

5.5.6.1.1 General

AES70 features in this area are organized around the block mechanism (see 5.4.3). The library feature is optional - devices need not implement any library-related objects or functions.

OcaLibrary shall define a mechanism for the handling of parameter value sets that are pre-stored in devices.

> NOTE: Such parameter sets have historically been given such names as "preset", "patch", "memory", or "scene".

AES70 defines two kinds of prestored parameter sets, as follows:

5.5.6.1.2 ParamSet

A **ParamSet** shall be a pre-stored set of property values for one particular class of block. Read-only properties (for example, Class ID) shall not be saved or restored by this mechanism.

The act of installing these values into an instance of the block is termed *applying the **ParamSet** to the block*. A stored definition which specifies the application of a particular **ParamSet** to a particular block instance is termed a **ParamSet** assignment.

A **ParamSet** shall not be required to contain values for all of the properties in its target block. For example, a **ParamSet** may contain only one value. Applying a **ParamSet** to a block shall change only the property values contained in that **ParamSet**, and shall leave other properties unchanged.

The block class for which a **ParamSet** is defined is termed the **ParamSet**'s *target block class*. A single **ParamSet** may be applied to any number of blocks, as long as all of the blocks are instances of the **ParamSet**'s target block class.

The binary format of a **ParamSet** is device-dependent and is not specified within this standard. Within AES70, a **ParamSet** shall be treated as a BLOB. This standard includes primitives for upload, download, creation, and management of **ParamSet**s, but these primitives do not include functions that examine those **ParamSet**s.

> EXAMPLE Suppose a mixing console defined a block class named **InChannel** to represent one input channel. Thus, a 32-input console would have 32 instances of this class. The console could define one or more channel **ParamSets** for **InChannel**. Any of these **ParamSets** could be applied to any instance of **InChannel**.

5.5.6.1.3 Patch

A Patch shall comprise a set of **ParamSet** Assignments. The act of executing all the assignments in a Patch is termed *applying the Patch to the device*. Applying a Patch to a device affects only the parameter values included in that Patch's **ParamSet**s; other parameters shall remain unaffected.

> NOTE The term "patch" inherits from the similar concept in the MIDI device control protocol. A MIDI "patch" is a set of MIDI commands that configure a device into a particular state.

5.5.6.2 OcaLibrary structure

A collection of **ParamSets** is termed a *ParamSet library*. A collection of Patches is termed a *Patch library*. The number of **ParamSet** libraries or Patch libraries that a device may implement is unlimited.

> NOTE A **Patch** is not the same as a **ParamSet** library, because a **ParamSet** library is a collection of **ParamSets**, but a **Patch** is a collection of **ParamSet** assignments.

Each library in a device shall be represented by an **OcaLibrary** instance. All of these instances, both for **ParamSet** and **Patch** libraries, shall be collected by the **OcaLibraryManager** object.

5.5.6.3 Creating ParamSets

AES70 defines two ways of creating **ParamSets** and storing them in the device:

1. A controller may download a **ParamSet** to a designated library in the device, OR
2. The controller may request that an **OcaBlock** object in the device save all of its property values as a **ParamSet** in a particular library in that same device. This "snapshot" action shall capture the values of all the properties of all the objects inside the block and save them as a **ParamSet** in the designated library.

5.5.7 OcaMediaClock

OcaMediaClock shall describe a particular internal or external media clock which the device uses. It shall include features for specifying parameters including clock sources, frequencies, rates, and lock states.

A device may define any number of media clocks; each one shall be represented by its own instance of **OcaMediaClock**. All of these instances shall be collected by the Media Clock Manager (class **OcaMediaClockManager**).

If a device has no network-controllable clocking features, it need not instantiate **OcaMediaClockManager** or **OcaMediaClock**.

5.5.8 OcaEventHandler

See clause 6.

5.6 Manager classes

Manager classes are listed in table 4.

Table 4 - Manager classes and standard object numbers

ONo	Class name	Function(s)
1	**OcaDeviceManager**	Manages information relevant to the whole device - including model and serial number, device name and role, overall operating state, and device update lock.
2	**OcaSecurityManager**	Manages security keys.
3	**OcaFirmwareManager**	Performs firmware updating.
4	**OcaSubscriptionManager**	Manages subscriptions, the constructs by which devices inform controllers of significant events. See 5.7.
5	**OcaPowerManager**	Manages device power state, including multiple power supplies, battery supplies.
6	**OcaNetworkManager**	Manages the control and media transport network(s) to which the device is connected.
7	**OcaMediaClockManager**	Manages media clock selection and control.
8	**OcaLibraryManager**	Manages device patches and presets, the elements that handle pre-stored device configurations and parameter settings. See 5.5.6.
9	**OcaAudioProcessingManager**	Manages global audio processing parameters.
10	**OcaDeviceTimeManager**	Provides access to device time-of-day clock.
100	**OcaBlock**	Root block.

NOTE Some manager objects will be required for required for all devices, others may be optional - see *AES70-1*.

5.7 Standard Object Numbers (ONo)

Every device shall assign standard object numbers to certain specific objects. These object numbers are given in table 4.

5.8 Object text identification

Every AES70 object shall include a read-only text property named Role, which shall state the role of the Worker in the device; for example, "Preamp Gain".

Additionally, every Worker object and Agent object shall include a writable text property named Label that controllers may use to record the function of the object in the application context; for example, "Elvis Vocal Gain".

5.9 Constructing objects

5.9.1 General

By definition, fully-configurable devices shall allow controllers to construct Worker and Agent objects. These functions shall be provided by the **OcaBlock** methods:

For a fully-configurable device, a controller may call **ConstructMember** to construct an object. When called, **ConstructMember** shall construct the object based on object-specific parameters. These parameters are termed *construction parameters*, and are defined in *AES70-2* for each class. When a controller calls **ConstructMember**, it may pass values for some or all of the construction parameters. *AES70-2* provides default values for parameters not specified.

For example, the class **OcaSwitch** defines a general *n*-position switch with text labels for each position. At construction time, the controller may specify the number of positions and the label text for each position. The default is a two-position switch with blank position labels.

In fully-configurable devices, controllers shall be able to build blocks, populate them with objects, and connect signal paths among those objects. There shall be two options for doing this, as follows:

1. A controller may call **ConstructMember** to construct the block, then call **ConstructMember** again to construct objects within the new block. It may then call **AddSignalPath** as needed to define signal flows within the block. or;
2. A controller may call **ConstructMemberUsingFactory**, referencing a particular block factory object, to construct a block with a predefined set of objects and signal flows as defined by the block factory object being used.

5.9.2 Block Factories

Each block factory shall be an instance of class **OcaBlockFactory**, defined in *AES70-2*. A block factory shall be an object whose function shall be to construct a specific kind of block, with a predefined set of objects and signal paths. A block factory shall reside in the same device for which it creates blocks.

There shall be three options for creating block factories. Some or all of these options may be implemented, depending on device configurability, as follows:

1. Block factories may be defined by the device's firmware.
2. If the device is pluggable, block factories may be defined at the time of device setup.
3. If the device is fully-configurable, the controller may create block factories using AES70 commands.

5.10 Deleting objects

In some devices, an object may be deleted using the **DeleteObject** method of the block where it resides:

1. In static, pluggable, and partially configurable devices, no object may be deleted.

2. In fully-configurable devices, any Worker (including block) or Agent may be deleted.

When a Worker is deleted, all signal paths that connect to it shall be deleted. When a block is deleted, all objects within it shall be deleted.

6 Events and subscriptions

6.1 Subscriptions, events, emitters and notifications

6.1.1 General

Subscription means a persistent relationship between two objects, in which one object sends update messages to the other object automatically, when certain specified conditions in the device occur. Such conditions are termed *events*, the transmitting object is termed an *emitter*, and the transmitted message is termed a *notification*. A notification type shall be specifically defined for each type of event. An object that sends a notification is said to *raise the related event*.

An emitter may implement more than one type of event, and may therefore emit more than one type of notification.

6.1.2 Reliable or fast subscriptions

Subscriptions shall be created by the Subscription Manager in response to controller calls to the **`AddSubscription`** method of the Subscription Manager. A subscription may be created in either of two forms: *reliable* or *fast*. A reliable subscription shall send notifications via the Reliable Message Delivery service; a fast subscription shall send notifications via the Fast Message Delivery service. Message delivery service classes are described in 4.3.2.

6.1.3 Subscription deletion

Subscriptions shall persist until the controller deletes them or until they are abandoned. An abandoned subscription is a subscription whose subscriber no longer appears in the network. In most implementations, such detection of subscriber failure should be done by using keep-alive messages - see 11.2.1.

When a device detects a subscriber failure, it shall delete all subscriptions which were made by the failed subscriber.

6.1.4 Subscription event handler

The element which receives a notification is termed an *event handler*. An event handler is an **`OnEvent`** method of an instance of the Agent class **`OcaEventHandler`**. The object that owns the event handler is termed the *subscriber*.

When a subscribed event occurs, the emitter shall transmit a notification to the subscriber's event handler. This notification is equivalent to a call to the **`OnEvent`** method of the subscriber. The notification shall contain information which allows the event handler to perform the appropriate processing.

AES70 does not limit the number of subscribers to an event, nor the number of subscriptions in which an event handler may participate.

> NOTE Although there is no limit on numbers of subscribers or subscriptions, in practice there will be implementation limits that vary from device to device.

6.2 The PropertyChanged event

The root class **`OcaRoot`** shall define an event named **`PropertyChanged`** which, when used, shall be raised whenever any of its object's property values changes.

`PropertyChanged` subscriptions may be used for such purposes as:

- Monitoring signal-induced object state changes;
- Monitoring overall device state;
- Updating controllers to match user adjustments to front-panel controls;
- In multicontroller systems, updating parameter status among the various controllers.

An object shall raise the **`PropertyChanged`** event whenever any of its properties changes. Data returned to the subscriber shall identify which of the object's properties has changed value.

NOTE 2 Through the class inheritance mechanism, the **`PropertyChanged`** event is defined for every AES70 class. Thus, a controller can monitor the property values of an object simply by subscribing to its **`PropertyChanged`** event. Since all of a device's controllable parameters reside in its properties, the **`PropertyChanged`** event gives complete access to the device's controllable operating state.

6.3 Use of numeric observers

The `OcaNumericObserver` class is defined in *AES70-2*. Each **`OcaNumericObserver`** object shall monitor a specified parameter value and raise an event named *observation* whenever the observation criteria are met.

Where required, a numeric observer may be created in the device, assigned relevant criteria - numeric test, periodic repetition rate, or both - and bound to the property to be observed. A controller may then subscribe to the numeric observer's observation event. Subsequently, whenever conditions meet the numeric observer's criteria, the numeric observer shall transmit an observation notification to the controller's event handler.

NOTE 1 Numeric observers may be created at time of device manufacture, or, for dynamic devices, by controllers at a later time.

NOTE 2 Numeric observers will commonly be used in conjunction with sensor objects (for example, audio level sensors), but can be used equally with any property of any object. For example, a numeric observer may be defined that emits a notification whenever someone raises the gain of a power amplifier above some threshold level.

Figure 9 shows an example of a numeric observer that implements a signal-present light in a controller.

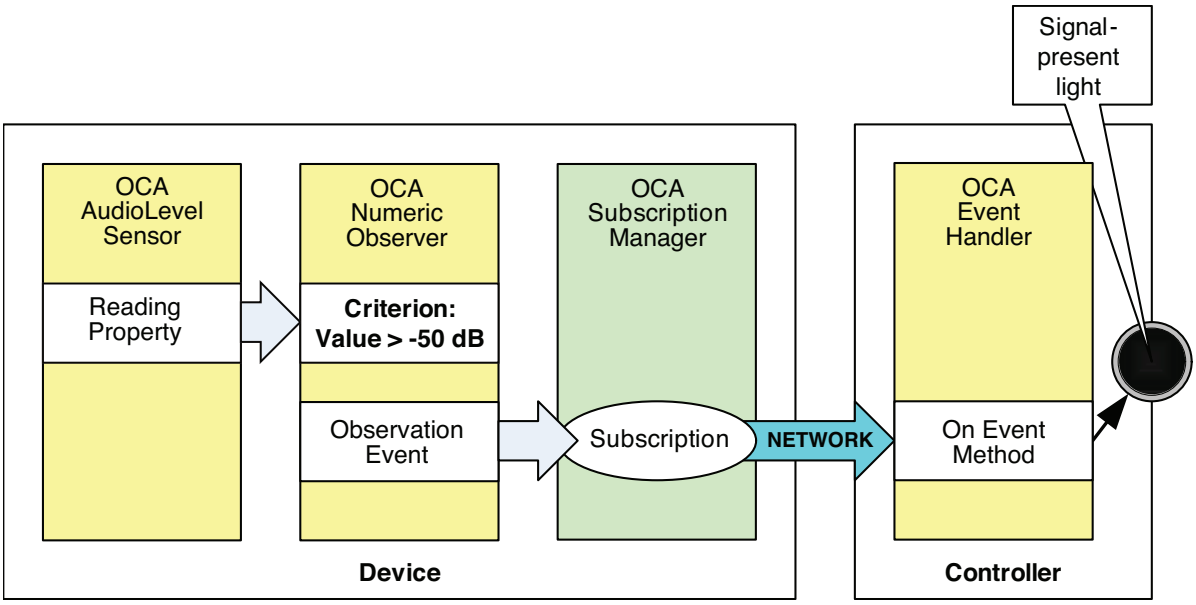


Figure 9 - Using a numeric observer to implement a signal-present light

7 Networking

7.1 General

In AES70, the concept of network shall be generalized: a network may support control and monitoring, or streaming media transport, or both. For networks that provide media transport, the control interface shall be generic, in order to allow AES70 to work with media networks of various types.

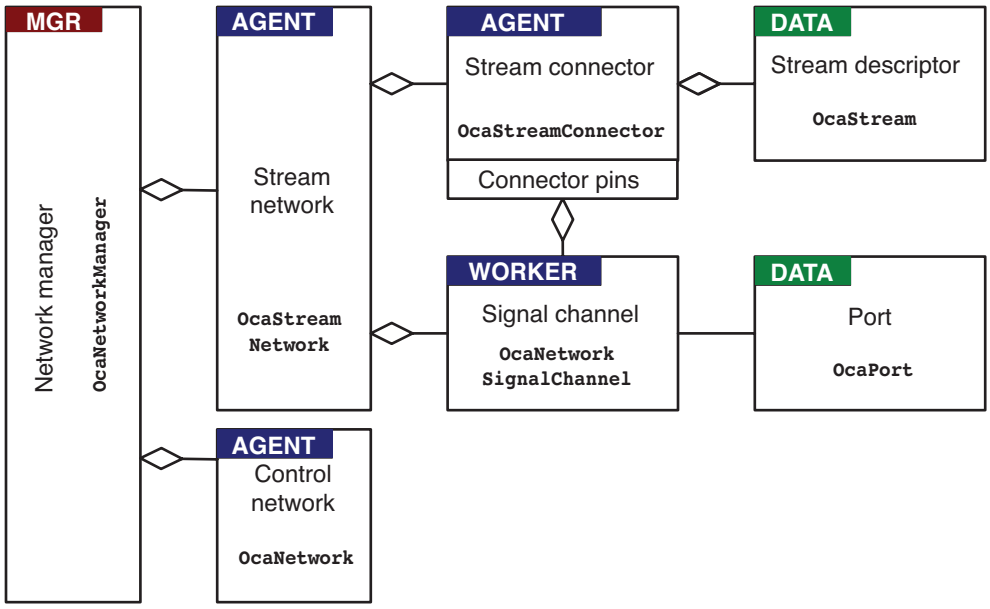
AES70 shall support devices that belong to more than one network at a time. The multiple networks may all be of the same type, or they may be of assorted types.

AES70 classes involved with networks are listed in table 5.

Table 5 - Networking classes

Name	Type	Function(s)
OcaNetworkManager	Manager	Collects all networks to which the device belongs
OcaNetwork	Agent	Represents a network that carries control traffic
OcaStreamNetwork	Agent	Represents a network that streams media and optionally carries control traffic
OcaStreamConnector	Agent	Represents a connection point for inbound or outbound media streams (see below)
OcaStream	Datatype	Represents an inbound or outbound media stream (see below)
OcaNetworkSignalChannel	Worker	Allows connection of an external inbound or outbound signal to one or more signal paths within the device.

Figure 10, below, shows the relationships between the networking classes of AES70.



NOTE in this figure, a diamond-headed arrow represents a collection relationship, with the objects at the plain ends being collected by the object at the diamond-headed end.

Figure 10 - Network object relationships

This figure shows both an **OcaStreamNetwork** object and an **OcaNetwork** object. The **OcaStreamNetwork** object shall be capable of managing both signal transport and AES70 control traffic. The **OcaNetwork** object shall be capable of managing only AES70 control traffic. A device may use either or both of these classes.

- If AES70 is used to manage media transport connections, **OcaStreamNetwork** shall be used.
- "Control-only" devices with no media transport capability may use either **OcaStreamNetwork** or **OcaNetwork**.
- New designs should use **OcaStreamNetwork**.
- If a device belongs to multiple networks, multiple instances of **OcaStreamNetwork** should be used as needed.

7.2 Media Transport Connection Management

Through **OcaStreamNetwork** and its associated classes (figure 10), AES70 controllers shall be able to access media network connection management functions for setting up, monitoring, and tearing down media transport connections between devices.

7.3 OCA Adaptations

7.3.1 General

AES70 network classes shall form a foundation for managing connections of all kinds of media transport networks. However, it is recognized that specific adjustments may be required to support particular media transport networks. The specification of the AES70 configuration for a particular network type shall be termed an *AES70 adaptation*.

In general, AES70 adaptations shall contain rules for configuring and using AES70 networking classes, and may additionally define specific refinements (subclasses) of standard AES70 classes.

Some adaptations may require multiple **OcaStreamNetwork** instantiations for the same physical network. For example, an AES70 adaptation for a network that provided TCP/IP for control and monitoring, but used a different protocol suite for media transport, might entail two **OcaStreamNetwork** Agents, one for IP control traffic, and another for the media stream(s).

AES70 adaptations are outside the scope of this standard.

7.3.2 Kinds of Media Networks

The AES70 media transport management scheme shall support both *stream-based* and *channel-based* media networks.

- *Stream-based media network* shall denote a media network that collects signal channels into unidirectional groups called *streams.* In such networks, applications make inter-device connections on a stream by stream basis, and the number of channels in each stream may vary from stream to stream.
- *Channel-based media network* shall denote a media network in which applications make inter-device connections on a signal-channel by signal-channel basis. In the implementation of such a network, signal channels may or may not be grouped into streams, but, if present, such groupings will typically be managed automatically and will be invisible to applications.

AES70 shall place no restrictions on media stream formats, sampling rates, encodings, or other media data representations.

7.3.3 Examples

Examples of stream-based and channel-based connection management are in given annex C.

7.3.4 The OcaStreamNetwork Class

The media network management mechanisms of AES70 shall be organized around the **OcaStreamNetwork** class. **OcaStreamNetwork** objects shall store basic network access information and shall hold two key collections - *stream connectors* and *signal channels*. These are defined below.

> NOTE Most devices will require only a single **OcaStreamNetwork** object to handle all their media and control network traffic. Multiple **OcaStreamNetwork** objects shall be used in devices that belong to more than one network at a time.

Each **OcaStreamNetwork** instance shall provide the following features for the network it represents:

1. Functions to start up, pause, and shut down the interface for the network.
2. Standardized network status indication.
3. Ability to get and set the host name or ID by which the device advertises itself on the network.
4. Identification of the hardware link type (TCP/IP Ethernet, USB, etc.) of the network.
5. Identification of the software interface the network uses for input and output.
6. Identification of the control (if any) and media transport (if any) protocols the network uses.
7. Transmission performance and error statistics.
8. Collections of associated classes (see below).

The specific use of **OcaStreamNetwork** and its associated classes shall depend on whether the media network is stream-based or channel-based. In what follows, these two modes are described separately.

7.3.5 Mode 1: Stream-based Media Connection Management

7.3.5.1 General

For stream-based connections, a stream shall connect to a *stream connector*, channels within the stream shall connect to *connector pins*, and the connector pins shall connect to internal device signal paths. These concepts are described below.

7.3.5.2 Stream connectors

A stream connector shall be an instance of class **OcaStreamConnector**.

A stream connector shall be an Agent object that defines the binding of the device to an external media transport stream for the purpose of sending or receiving stream data.

The **OcaStreamConnector** object shall include methods which allow controllers to create and delete media stream connections. As well, the **OcaStreamConnector** object shall be capable of informing subscribing controllers of media stream connections that have been made or broken in response to requests from external devices and controllers.

The **OcaStreamConnector** object shall include a feature for identifying a media transport connection as secure. However, the specific implementation of media transport security shall be outside the scope of this standard.

A stream connector may be a source (output) connector or a sink (input) connector. Bidirectional connectors shall not be supported. Source connectors - but not sink connectors - may connect to multiple outbound streams.

For each stream connected to a stream connector, the connection parameters shall be recorded in a stream descriptor. See 7.3.5.3.

7.3.5.3 Stream Descriptors

A stream descriptor shall be a data element defined by the **OcaStream** datatype class; it shall hold the properties of a network media stream. Each stream connector shall collect a stream descriptor for each stream to

which it is connected. Thus, the **OcaStreamConnector** class has a collection relationship with **OcaStream** class. See figure 10.

A source (output) stream connector may connect to multiple streams, and may therefore hold multiple stream descriptors. A sink (input) stream connector may connect to at most one stream, and shall therefore hold at most one stream descriptor.

7.3.5.4 Connector pins

A stream connector is analogous to a traditional multipin signal connector. Continuing this analogy, each stream connector shall have a set of one or more *connector pins*. Each connector pin shall be a data element that maps a single signal from a connected stream to an **OcaNetworkSignalChannel** object. The port of the **OcaNetworkSignalChannel** object may in turn be connected to one or more internal signal paths inside the device, using the normal AES70 signal flow mechanism, described in 5.4.3.6.

7.3.5.5 Signal channels

A signal channel shall be an instance of class **OcaNetworkSignalChannel.**

A signal channel shall be a Worker object that provides the binding of a particular connector pin to a particular signal path or paths inside the device. Each signal channel shall contain an **OcaPort** property to which internal device signal paths may be connected.

A signal channel may be a source (output) channel or a sink (input) channel. Bidirectional channels shall not be supported.

A source (output) signal channel may be associated with multiple connector pins in multiple connectors. A sink (input) signal channel shall be associated at most with one connector pin.

7.3.6 Mode 2: Channel-based media connection management

For channel-based connections, each inbound or outbound signal channel in the network shall connect to a signal channel object (7.3.5.5) in the device. The signal channel object shall be an instance of the **OcaNetworkSignalChannel** class. The signal channel object shall store the relevant connection information for the channel to which it connects.

An output channel may connect to multiple channels in the network; an input channel shall connect to at most one channel in the network. Internally, via its **OcaPort** each signal channel object may connect to intra-device signal paths according to the usual AES70 signal flow rules (5.4.3.6).

There shall be an **OcaStreamNetwork** object to describe the network; this object shall collect all the **OcaNetworkSignalChannel** objects.

These relationships are shown in figure 11.

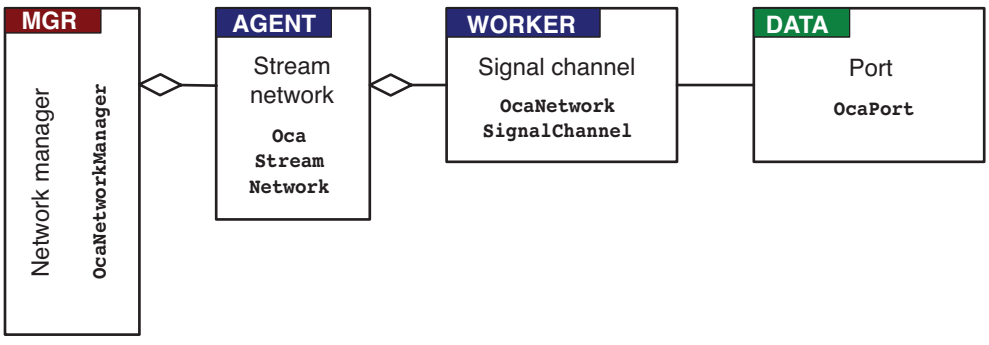


Figure 11 - Channel-based classes

8 Sessions

AES70 protocols are session-oriented, which implies the following:

1. Requests and responses shall be correlated pairs.
2. Objects shall have permanent application relationships with other AES70 objects.
3. Prompt discovery of device failure shall be required.
4. Devices shall be able to report changing parameters to subscribing controllers on a regular and continuing basis (for example, signal level).
5. Controller-to-device relationships shall survive or be deleted in a predictable manner when transport interruptions occur.

NOTE Networks may contain thousands of AES70 devices. It may not be practical for a single central controller to have direct control over hundreds or thousands of devices, maintaining a separate transport connection with each one. In such cases, indirect control using multiple-controller hierarchies may be implemented, with successive levels aggregating control functionality in ways appropriate to the applications. Aggregation features - notably the **`OcaGrouper`** class (see 5.5.3) - will aid such implementations.

9 Security

The aim of AES70 is to support networks capable of operating at levels of security sufficient to satisfy:

- international regulations governing emergency evacuation systems.
- commercial and government data security requirements.
- public media and live performance data security requirements.

AES70 security depends on the communications protocol being used.

For example, in networks using TCP/IP communications and the AES70-3 protocol, security of the control data shall use the TCP/IP family's Transport Layer Security (TLS) protocol to provide authentication and encryption - see *AES70-3* for details.

Security of the control data may be enabled or disabled globally; that is, for the entire AES70 network. An AES70 network shall not operate with a mix of secure and insecure devices.

NOTE The AES70 specification excludes access control. Access control defines which devices, objects, object features, and object value ranges can be affected by each user. If access control is required, then it may be implemented in the application, and AES70 security may be used to secure the implementation.

10 Concurrency control

AES70 shall support applications that use multiple simultaneous controllers in which a particular object may receive directives from more than one controller.

In such applications, certain application control events require the exchange of more than one control message. Therefore the potential for a race condition exists. To prevent race conditions, AES70 devices shall support single-threading via a simple object-locking mechanism with the elements listed below.

An AES70 lock is designed to prevent all access to the object by anyone other than the lockholder.

The lock interface shall be defined in the **`OcaRoot`** class, and shall therefore be inherited by every other AES70 class. However, non-lockable objects may be defined, if desired, as described below.

AES70 locking rules are as follows:

1. The lock interface shall be defined for every AES70 object.

2. An object may be implemented as lockable or non-lockable. A non-lockable object shall return a not-implemented status (via a response message - see 4.3) in response to all locking-related commands.
3. A lockable object may be locked by at most one remote object (the *lockholder*) at a time.
4. A lock shall not survive a device reset.
5. Controller to device communication should be continuously monitored (see 11.2). When a controller's communication with a device fails for any reason, the device shall automatically remove all locks set by that controller.
6. It shall be possible to lock an entire device by locking its Device Manager object, provided that none of the device's other objects are locked.

NOTE AES70 does not include a full lock management function. For example, there is no deadlock detection, there are no non-exclusive locks, there are no multi-level locking schemes. Such functions, if needed, are left up to the application.

11 Reliability

11.1 General

The aim of AES70 is to support networks capable of operating at levels of reliability sufficient to satisfy:

- international regulations governing emergency evacuation systems.
- commercial and government uptime and robustness requirements.
- public media and live performance uptime and robustness requirements.

"Reliability" means the cumulative effect of Availability and Robustness.

11.2 Availability

11.2.1 Keep-alive messages

The availability of AES70 networks should be monitored by continuous device supervision, using periodic *keep-alive* messages defined as part of the network protocol specification. The nature of this message will depend on the particular network protocol being used.

Devices may also monitor themselves by means of local network protocol messages sent through the complete path from application to network and back.

11.2.2 Efficient reinitialization

In the event of errors and configuration changes, the protocol implementation should reinitialize the affected network devices quickly and efficiently.

11.3 Robustness

To support robust implementations, AES70 shall include mechanisms to confirm operation, and shall define fault-tolerant mechanisms that are resistant to PDU losses and device failures.

AES70 protocol implementations may use network-type-specific robustness mechanisms.

For example, when the protocol defined in AES70-3 (OCP.1) runs on Ethernet, its implementation can take advantage of spanning-tree protocols to increase resistance to network link failures.

12 Device reset

An AES70 device may support the *device reset* function to recover from catastrophic errors. A device reset shall have the effect of returning the affected device to the state it was in immediately following power-up. A device reset shall cause all of the device's initialization data (for example, routing information) to be deleted.

NOTE 1 It is expected that device resets will be used only in extreme situations.

A device reset shall be invoked by the device's receipt of a device-reset command. A device-reset command shall be a special message that includes a 128-bit security key (the *reset key*). The value of this key shall match a previously-stored value in the device. If it does not, the device-reset command shall have no effect.

Reset key values shall be set by a particular AES70 command, and the memory of key values shall not survive a power-up reset. If a device has not received a reset key value since its most recent power-up reset, it shall ignore all device-reset commands.

NOTE 2 Since a successful device-reset command causes a power-up reset, a device-reset command will cause the affected device to forget its reset key.

Device-reset commands shall be sent via the Fast message delivery service, and may be multicast where the protocol in use permits.

NOTE 3 When AES70 security is not being used, the reset key mechanism is insecure, since the key may be reset at any time. Full security of the reset mechanism is only available when overall AES70 security is enabled. For insecure implementations, the reset-key mechanism is intended to reduce the probability of casual system resets.

13 Firmware and software upgrade

13.1 General

AES70 shall define mechanisms for reliable upgrading of device firmware or software over the network. The implementation of such mechanisms shall be optional. In what follows, the term *updater* shall refer to the AES70 controller that is driving the firmware update process.

When AES70 network upgrading is implemented, then:

1. Upgrading shall be implemented in a way which ensures the device can recover from a failed update. Thus, the device must have a protected bootup downloader that continues to function even when the device's program memory contains an invalid image, or no image.

2. Security of the firmware or software upgrade shall be handled through the normal security mechanism for control data - see clause 9.

3. Implementers of AES70-compliant firmware upgrade mechanisms should take care to ensure that firmware upgrades are properly signed and controlled. The specifics of such controls are outside the scope of AES70.

13.2 Update Types

AES70 shall support devices with multiple firmware components per device, and shall allow devices to support any of the four update types listed in table 6

Table 6 - AES70 update types

Update type	Description
1	Fully transaction-based updates, in which newly received component firmware images shall be placed into service only when all component image loads have succeeded. If any component update fails, the device shall revert to all of its old images.
2	Partly transaction-based updates, in which each newly received component firmware image shall be placed into service immediately after its particular load has succeeded. If the component's image load does not succeed, the device shall revert to the component's previous image. NOTE This method can cause a device to contain a mix of old and new images at the end of a partially-successful update process. If a device's implementation is not tolerant of this situation, the device may be rendered nonfunctional and non-updateable.
3	Guarded non-transaction-based updates, in which incoming component firmware images shall immediately overwrite current firmware. If an image load fails, the device shall reload failsafe (sometimes called golden) images from internal read-only memory; these images shall provide sufficient function for manual retry of the update and/or reversion to pre-update software versions.
4	Unguarded non-transaction-based updates in which incoming component firmware images shall immediately overwrite current firmware, with no failsafe mechanism to recover from failed uploads. NOTE Failed unguarded non-transaction-based uploads may render devices nonfunctional and non-updateable.

13.3 Update Modes

AES70 shall support two modes by which firmware image data is updated, as follows:

1. Active updating, in which the updater shall call specific methods in the device to upload firmware image data into said device;

2. Passive updating, in which the device shall download firmware image data from a source specified by the updater.

13.4 Update mechanisms

13.4.1 General

All firmware updating shall be managed by the **`OcaFirmwareManager`** object, which is a mandatory Manager in every AES70 device.

The active updating process differs from the passive updating process. Either process can support any of the four update types described in 13.2. The choice of update type shall be a device implementation option.

13.4.2 Active Updating

The active update of a device shall proceed as set out in table 7.

Table 7 - Active updates

1.		The updater shall call the **`OcaFirmwareManager.StartUpdateProcess()`** method.
2.		For each firmware component image to be updated, the updater shall:
	a.	Call the **`OcaFirmwareManager.BeginActiveImageUpdate()`** method to start the component update.
	b.	Call the **`OcaFirmwareManager.AddImage()`** method as many times as necessary to upload the complete component firmware image to the device.
	c.	Call the **`OcaFirmwareManager.VerifyImage()`** method, which causes the device to verify the integrity of the uploaded component firmware image.
	d.	Call the **`OcaFirmwareManager.EndActiveImageUpdate()`** method, which causes the device to complete the update of a particular component.
3.		The updater shall call the **`OcaFirmwareManager.EndUpdateProcess()`** method, which shall cause the device to complete the update process.

In the case of fully-transaction-based updates, the device shall place the uploaded images into operation in step 3, but only if all uploads have succeeded.

In the case of partly-transaction-based updates, the device shall place each uploaded image into operation in step 2d, but only if the image verification has succeeded.

In the case of guarded and unguarded non-transaction-based updates, the device shall place the segments of each uploaded image into operation at the conclusion of step 2c.

13.4.3 Passive Updating

The passive update of a device shall proceed as set out in table 8.

Table 8 - Passive updates

1.		The updater shall call the **`OcaFirmwareManager.StartUpdateProcess()`** method.
2.		For each firmware component image to be updated:
	a.	The updater shall call the **`OcaFirmwareManager.BeginPassiveComponentUpdate()`** method, passing the hostname of a network fileserver and the filename of the component firmware image the device shall download from the fileserver.
	b.	The device shall download the specified firmware image file.
3.		When all image files have been downloaded, the updater shall call the **`OcaFirmwareManager.EndUpdateProcess()`** method, which shall cause the device to complete the update process.

In the case of fully-transaction-based updates, the device shall place the uploaded images into operation in step 3, but only if all uploads have succeeded.

In the case of partly-transaction-based updates, the device shall place each uploaded image into operation at the end of step 2b.

In the case of guarded and unguarded non-transaction-based updates, the device shall place each uploaded image into operation at the end of step 2b.

Annex A (informative) - Actuator example

A.1 General

To see how classes are generally constructed, we examine **OcaGain** in more detail. **OcaGain** is an actuator, but its general way of working is the same for all classes; workers, managers, or agents.

OcaGain's class tree inheritance is shown in figure A.1. Note that the topmost three classes - **OcaRoot**, **OcaWorker**, and **OcaActuator** - are abstract classes that will never be instantiated in isolation. They are present in the class tree to express certain common characteristics of objects, workers, and actuators, respectively.

NOTE In this figure, an open-headed arrow represents an inheritance relationship, with an object at the plain end being a child of the object at the arrowhead end.

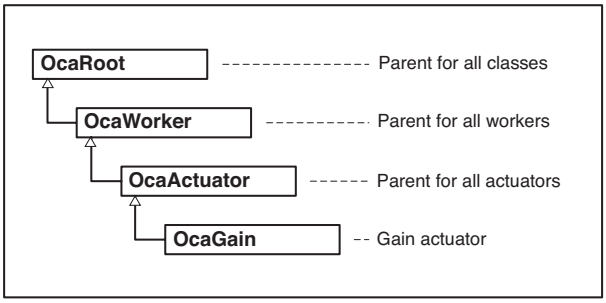


Figure A.1 - OcaGain lineage

A.2 Properties, Methods, and Events

OcaGain's properties, methods, and events are a combination of its own specific elements and elements inherited from its ancestors, **OcaActuator**, **OcaWorker**, and **OcaRoot**.

The complete set of properties is given in table A.1. These properties are accessed via method calls, shown in table A.2. Events that may be raised by **OcaGain** are shown in table A.3.

As the tables show, the repertoire of method calls is relatively large - larger than simple devices may need. AES70 defines a "not implemented" status value which objects may return for methods or method options not implemented in the device at hand.

Table A.1 - OcaGain properties

Name	ID	Datatype	Access	Description	Defined in
Object Number	01p03	ONo	RO	Object number of OcaGain instance	**OcaGain**
Lockable	01p04	Boolean	RO	True if object can be locked.	**OcaRoot**
Role	01p05	String	RO	Role of object in device, for example, "Channel 1 Gain"	**OcaRoot**
Enabled	02p03	Boolean	RW	True if object is enabled, false if object is disabled or the property has no effect	**OcaWorker**
Ports	02p04	Array of structures	DD	Collection of input and output ports that this object has	**OcaWorker**
Label	02p05	String	RW	Purpose of object in system, for example, "Elvis Vocal Gain"	**OcaWorker**
Owner	02p06	ONo	RO	Object number of containing block	**OcaWorker**
ClassID	04p01	Class ID	RO	Class ID of OcaGain class	**OcaGain**
Class Version	04p02	Integer	RO	Version number of OcaGain class	**OcaGain**
Gain	04p03	Float	RW	The gain value in dB	**OcaGain**

NOTE: In the Access column of table A.1, "RO" = read-only; "RW" = read-and-write, "DD" = device dependent

Table A.2 - OcaGain methods

Name	ID	Description	Defined in
GetClassIdentification()	01m01	Returns ClassID and version	**OcaRoot**
GetLockable()	01m02	Returns value of Lockable property	**OcaRoot**
Lock()	01m03	Locks the object	**OcaRoot**
Unlock()	01m04	Unlocks the object	**OcaRoot**
GetRole()	01m05	Returns value of Role property	**OcaRoot**
GetEnabled()	02m01	Returns value of Enabled property	**OcaWorker**
SetEnabled(...)	02m02	Sets value of Enabled property	**OcaWorker**
AddPort(...)	02m03	Adds a signal port to the object	**OcaWorker**
DeletePort(...)	02m04	Deletes a signal port from the object	**OcaWorker**
GetPorts()	02m05	Returns list of the object's signal ports	**OcaWorker**
GetPortName()	02m06	Returns name of a signal port	**OcaWorker**
SetPortName(...)	02m07	Sets name of a signal port	**OcaWorker**
GetLabel()	02m08	Returns value of Label property	**OcaWorker**
SetLabel(...)	02m09	Sets value of Label property	**OcaWorker**
GetOwner()	02m10	Returns ONo of containing block	**OcaWorker**
GetGain()	04m01	Returns value of Gain property	**OcaGain**
SetGain()	04m02	Sets value of Gain property	**OcaGain**

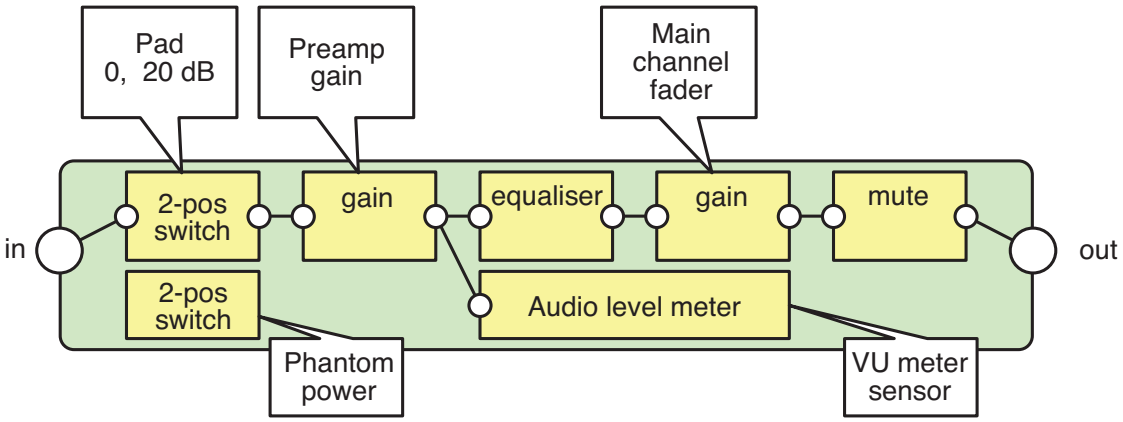
Table A.3 - OcaGain event

Name	ID	Description	Defined in
PropertyChanged(...)	01e01	Raised when a property of the object changes value	**OcaRoot**

Annex B (informative) - Block examples

B.1 Simple microphone channel

Figure B.1 shows a signal flow diagram for a typical microphone channel of a mixer.



NOTE The illustrations in this document use small circles for Worker ports, large circles for block ports, and simple lines for signal paths. Blocks are shown with rounded corners, other objects with square corners.

Figure B.1 - Simple microphone channel signal flow

B.2 Two-channel microphone mixer

Figure B.2 illustrates the use of blocks in larger assemblies. The microphone channel block of B.1 is replicated and combined with another gain control to make a simple microphone mixer.

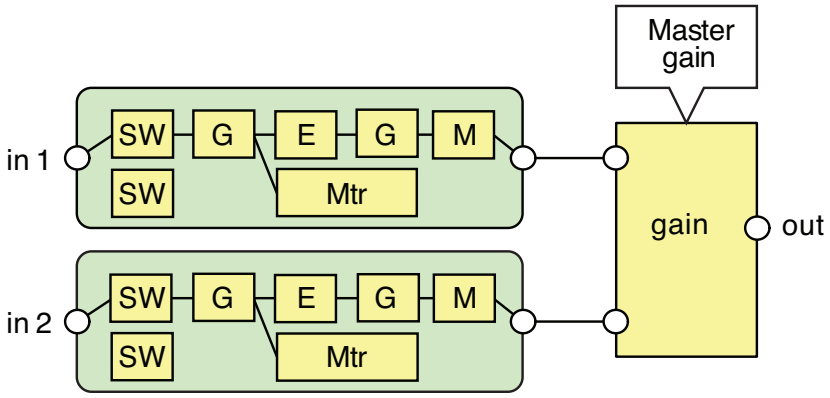


Figure B.2 - Two-channel microphone mixer

In this case, the master gain object has been given two input ports in order to represent the mixing function.

The reader might notice that there is no explicit object for the mix bus and summing amplifier. For this simple mixer, the mix bus and summing amplifier do not have any parameters that are remotely controllable. Therefore, they do not have a corresponding AES70 object. More sophisticated mixers might have control and/or monitoring functions associated with the summing subsystem. In that case, the signal flow might include an explicit summing point.

B.3 Mixer using nested blocks

This example considers the equalizer object shown in the microphone channel model in Figure B.1. A typical microphone channel equalizer might contain a high-pass filter section, followed by three parametric equalizer sections.

AES70 defines object classes that can be used to model a high-pass filter (HPF) or a parametric equalizer. To model our typical microphone channel equalizer, we could use one **`OcaFilterClassical`** instance, followed by three **`OcaFilterParametric`** instances.

These could simply be added to the microphone channel in sequence - see figure B.3.

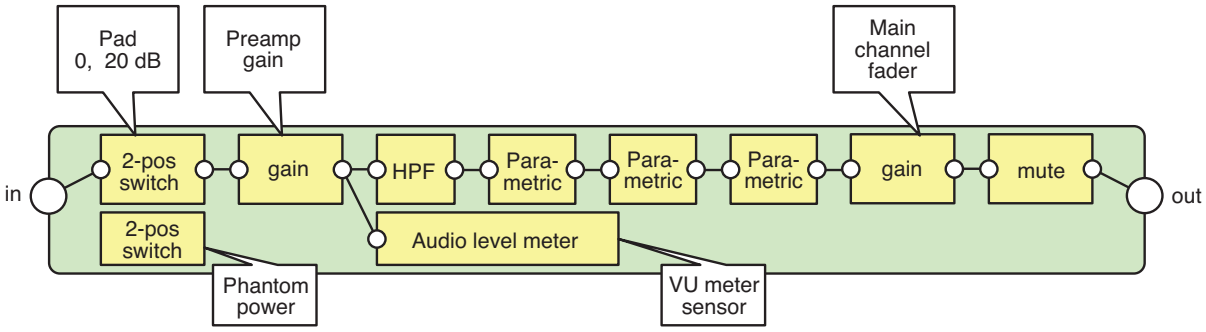


Figure B.3 - Mic channel with EQ sections inline

Alternatively, we could define a block named, say, **`MicEqualizer`** that contained all the equalizer objects and nest it inside the microphone channel block - see figure B.4.

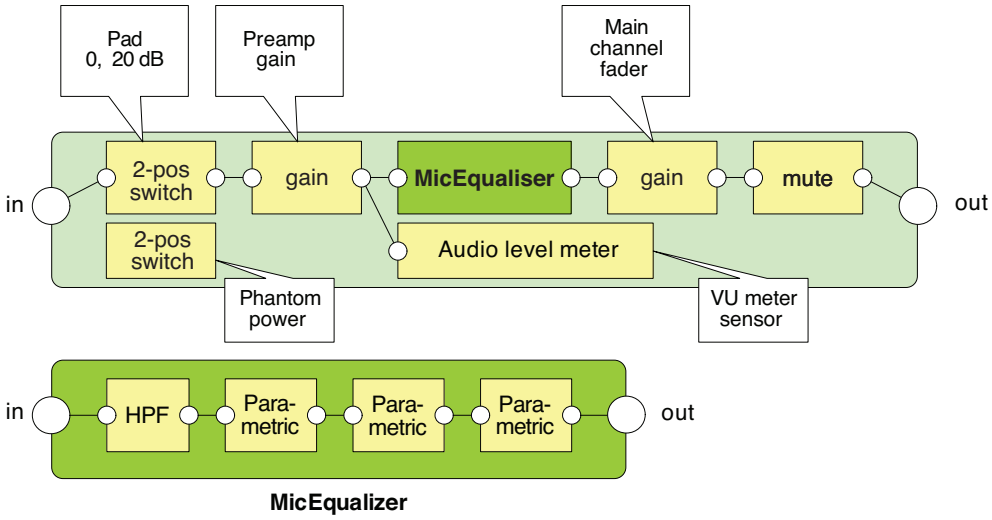


Figure B.4 - Microphone channel with MicEqualizer block

This arrangement will be easier to use in reconfigurable devices. It might also make controller implementation simpler, particularly if the same equalizer were used in various parts of the device, or in other products.

Annex C (informative) - Network connection management examples

C.1 Stream-based connection examples

C.1.1 General

The foregoing rules allow for a number of stream-based connection scenarios. Examples of these are illustrated here. These examples are not based on any particular transport network architecture; they are abstract cases to demonstrate the range of stream-based connections that AES70 is capable of managing.

C.1.2 Scenario 1: Input

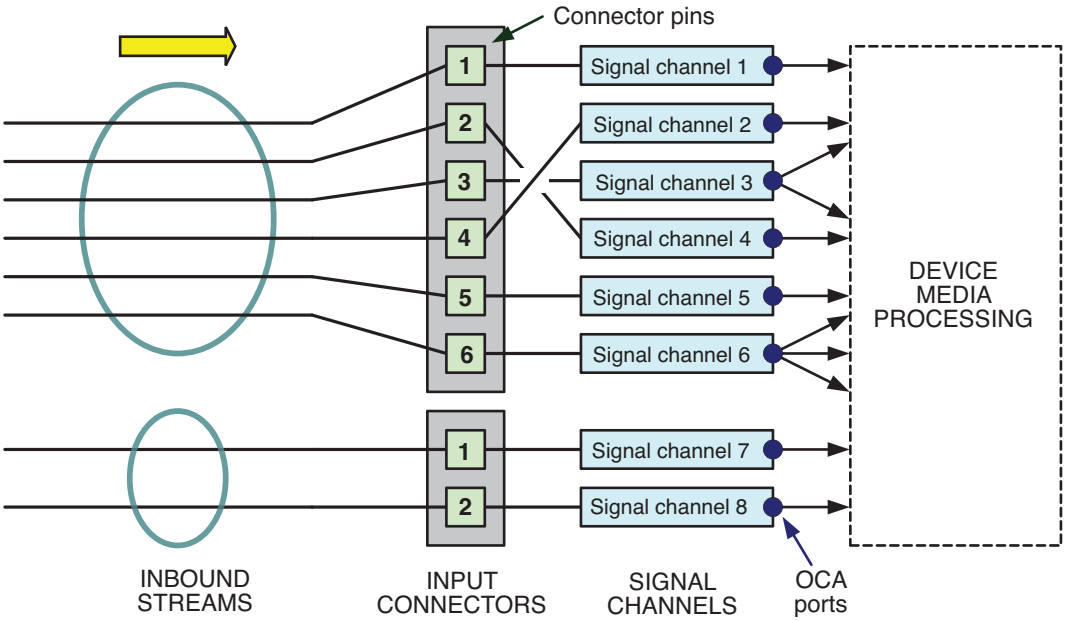


Figure C.1 - Stream-based input example

In this scenario, two inbound streams are connected to two connectors and thence to 8 signal channels. AES70 does not restrict the numbers or sizes of input connectors a device may have.

The device side of the connector shows some input remapping, as pins 2 and 4 of the first connector are mapped to signal channels 4 and 2, respectively. The device side of the signal channels shows further remapping, as is possible with the AES70 signal flow mechanism.

C.1.3 Scenario 2: Simple output

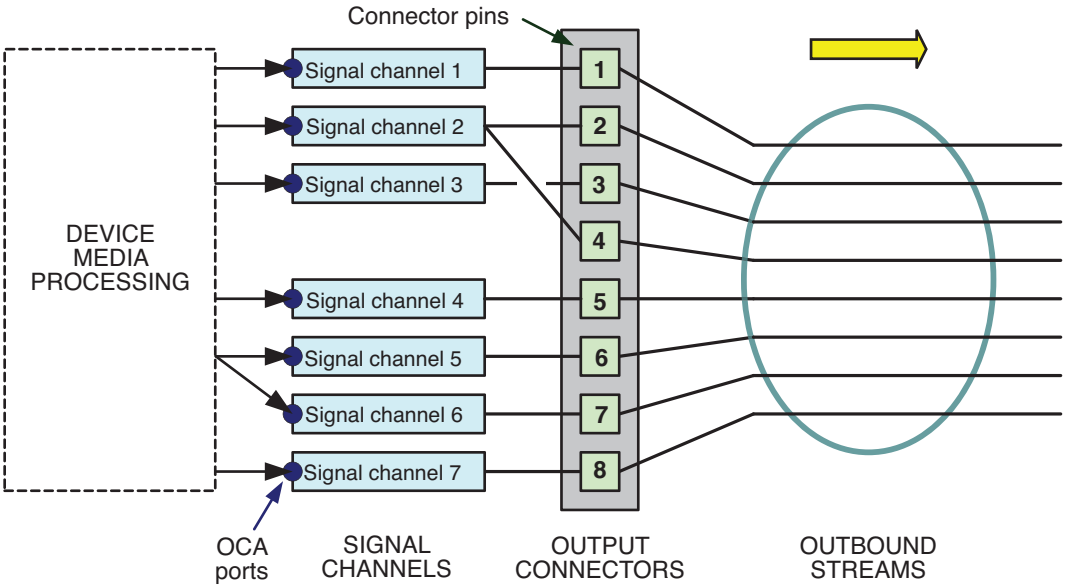


Figure C.2 - Simple stream-based output example

In this scenario, an 8-channel outbound stream is generated by 7 signal channels. Signal channel 2 drives two connector pins. The device side of the signal channels shows further remapping, as is possible with the AES70 signal flow mechanism.

C.1.4 Scenario 3: Multiple output connectors

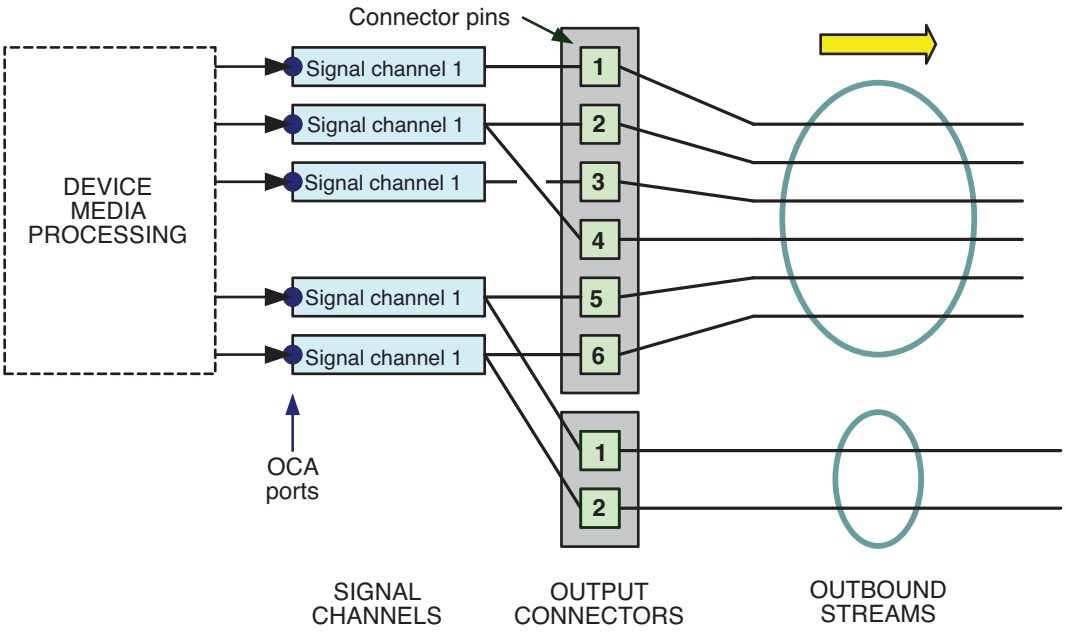


Figure C.3 - Multiple stream-based output connector example

In this scenario, two of the output signal channels drive multiple output connectors to yield a two-stream output capability.

C.1.5 Scenario 4: Multiple output streams from one connector

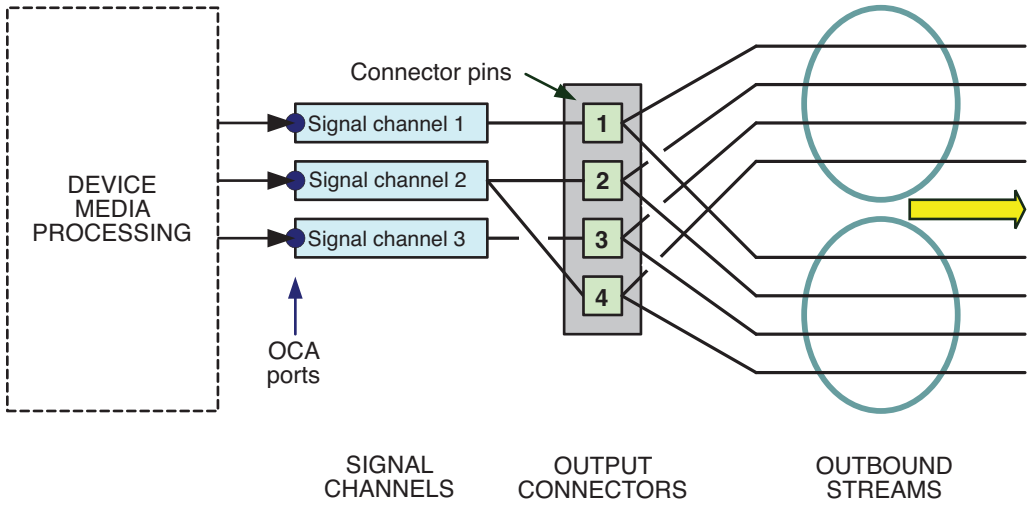


Figure C.4 - Multiple output streams from one connector

In this scenario, two identical outbound streams are driven by a single output connector.

C.2 Channel-based connection examples

C.2.1 Scenario 5: Input

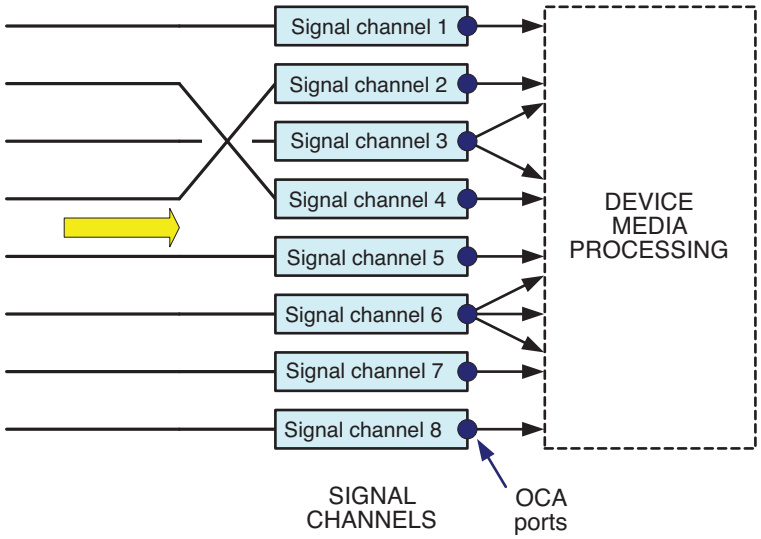


Figure C.5 - Channel-based input

In this scenario, eight inbound channels are connected to eight signal channels. The device side of the signal channels shows remapping, as is possible with the AES70 signal flow mechanism.

In contrast to the stream-based mode, signal channels in channel-based mode are not linked to connector pins. Instead, they are directly connected to network signal channels. The **`OcaNetworkSignalChannel`** class contains properties for identifying and describing the connected network signal channels.

C.2.2 Scenario 6: Output

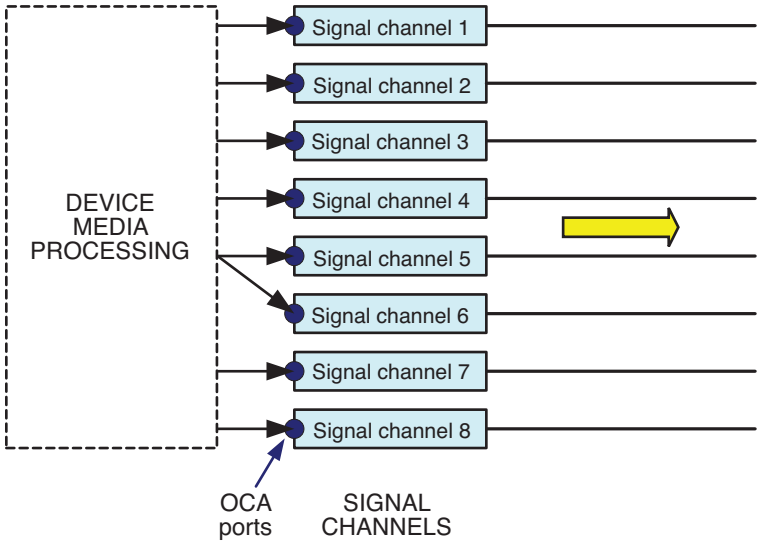


Figure C.6 - Channel-based output

In this scenario, eight output channels are generated by eight signal channels. The device side of the signal channels shows further remapping, as is possible with the AES70 signal flow mechanism.

Annex D (informative) - Other Media Network Control Standards

This Annex lists other media network control standards. The listed documents may be useful for comparative study, and for development of interoperability schemes.

D.1 AES64

AES64 is a previously-published control standard from the AES:

AES64-2012: *AES standard for audio applications of networks - Command, control, and connection management for integrated media*. Audio Engineering Society, New York, NY., US.

Online: http://www.aes.org/publications/standards/

D.2 SMPTE ST 2071 - Media Device Control

This standard describes a comprehensive control suite from the Society of Motion Picture and Television Engineers (SMPTE). Four documents, as follows:

SMPTE ST 2071-1:2014 *Media Device Control Framework (MDCF)*

SMPTE ST 2071-2:2014 *Media Device Control Protocol (MDCP)*

SMPTE ST 2071-3:2014 *Media Device Control Discovery (MDCD)*

SMPTE ST 2071-4:201x *Media Device Control Capability Interface Repository*

Documents are available via the SMPTE Digital Library at http://library.smpte.org/.

D.3 Architecture for Control Networks (ACN)

ACN is an older control protocol standard that originated in the theatrical lighting industry. Its definition is flexible, and it has been used by some audio manufacturers in recent years.

American National Standards Institute. "*E1-17: Architecture for Control Networks*".

Package of 17 documents plus supporting files. Online: http://webstore.ansi.org

D.4 Open Sound Control (OSC)

OSC is an open-source character-based protocol originally designed for electronic music applications. It has occasionally been adapted for professional audio use.

M. Wright. (2002, March), "*The Open Sound Control 1.0 Specification*".

Latest complete specification as of this writing. Version 1.1 is in development. Online: http://opensoundcontrol.org/spec-1_0

D.5 Ember+

Ember+ is an open-source audio device control protocol based on a programming standard called BER, for "basic encoding rules." It is not a formal standard, but has been used for audio device control by some manufacturers.

The specification is available at https://github.com/Lawo/ember-plus/wiki .