STANDARDS AND INFORMATION DOCUMENTS

AES70-1-2018
(Rev. AES70-1-2015)

AUDIO
ES
®
STANDARDS

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

AUDIO ENGINEERING SOCIETY, INC.
551 Fifth Avenue, New York, NY 10176. US.

AUDIO AES STANDARDS

The AES Standards Committee is the organization responsible for the standards program of the Audio Engineering Society. It publishes technical standards, information documents and technical reports. Working groups and task groups with a fully international membership are engaged in writing standards covering fields that include topics of specific relevance to professional audio. Membership of any AES standards working group is open to all individuals who are materially and directly affected by the documents that may be issued under the scope of that working group.

Complete information, including working group scopes and project status is available at http://www.aes.org/standards. Enquiries may be addressed to standards@aes.org

The AES Standards Committee is supported in part by those listed below who, as Standards Sustainers, make significant financial contribution to its operation.

Standards Sustainer Gold

NETFLIX Focusrite

Standards Sustainer Silver

LECTROSONICS 21ST CENTURY FOX Audio precision WAVES ADAMSON fulcrum ACOUSTIC ROSS LIVING LIVE! Eventide PreSonus DOLBY SENNHEISER THE TELOS ALLIANCE MUSIC TRIBE BOSE

Standards Sustainer

NHK NTi AUDIO MQA audio-technica LAWO CLAIR WEISS METRIC HALO AEA UDSOUND THAT Corporation NBCUniversal Control4 WorldCast Systems Fraunhofer IIS PRO PROAUDIOTECHNOLOGY MERGING AUDIO FOR THE NETWORKING AGE RAVENNA steinberg ARCHWAVE audio networking & streaming technologies ATTEROTECH HARMAN A SAMSUNG COMPANY KLIPPEL NEUTRIK dbx-tv XPERI

This list is current as of 2019/1/12

AES70-1-2018
Revision of AES70-1-2015

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

Published by
Audio Engineering Society, Inc.
Copyright © 2015, 2018 by the Audio Engineering Society

Abstract

AES70 defines a scalable control-protocol architecture for professional media networks. AES70 addresses device control and monitoring only; it does not define standards for streaming media transport. However, the Open Control Architecture (AES70) 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 AES70 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

2019-1-14 printing

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 9
3.1. AES70-compliant interface AES70 interface 9
3.2. Media device 9
3.3. Non-media device 9
3.4. AES70 device Device 9
3.5. Non-AES70 device 9
3.6. Device model 9
3.7. Controller 9
3.8. Control 9
3.9. Control protocol 9
3.10. Media stream 9
3.11. Open Control Protocol OCP 10
3.12. OCP.1 10
3.13. Protocol data unit PDU 10
3.14. Object Number ONo 10
3.15. Precision Time Protocol (PTP) 10
3.16. Program or AES70 Program 10
3.17. Task AES70 Task 10
3.18. Voltage-controlled amplifier VCA 10
3.19. Binary Large Object BLOB 10
3.20. Robustness 10
3.21. Availability 10
3.22. Alignment level 10
4. Top level design 10
4.1. General 10
4.2. Object orientation 11
4.2.1. General 11
4.2.2. Classes 11
4.2.3. Instantiation of classes 16
4.3. Messages 16
4.3.1. Basic mechanism 16
4.3.2. Message delivery services 17
5. Device model 17
5.1. Device configurability 17
5.2. Object addressing 18
5.3. Device model elements 19
5.3.1. General 19
5.3.2. Managers, Workers, and Agents 19
5.4. Worker classes 20
5.4.1. Actuators 20
5.4.2. Sensors 20
5.4.3. Blocks 21
5.4.4. Matrices 25

5.5. Agent classes 30
5.5.1. General 30
5.5.2. OcaGrouper 30
5.5.3. OcaRamper 35
5.5.4. OcaTask 36
5.5.5. OcaNumericObserver and OcaNumericObserverList 36
5.5.6. OcaLibrary 36
5.5.7. OcaMediaClock3 36
5.5.8. OcaTimeSource 36
5.5.9. OcaPhysicalPosition 37
5.5.10. OcaEventHandler 37
5.6. Network classes 37
5.7. Manager classes 37
5.8. Standard Object Numbers (ONo) 38
5.9. Object text identification 38
5.10. Constructing objects 38
5.10.1. General 38
5.10.2. Block Factories 39
5.11. Deleting objects 39
6. Events and subscriptions 39
6.1. Subscriptions, events, emitters and notifications 39
6.1.1. General 39
6.1.2. Reliable or Fast subscriptions 39
6.1.3. Lightweight subscriptions 40
6.1.4. Subscription deletion 40
6.1.5. Subscription aggregation 40
6.2. Subscription event handler 41
6.3. The PropertyChanged event 41
6.4. Use of numeric observers 42
7. Networking and connection management 42
7.1. General 42
7.2. Connection Management Versions 43
7.3. The CM3 object model 43
7.3.1. Classes 43
7.3.2. Structure 45
7.4. AES70 Adaptations (informative) 47
8. Time 47
8.1. Format 47
8.2. PTP Epoch 47
8.3. UTC time and the NTP time format 48
9. Physical Position Coordinate Systems 48
9.1. Robotic coordinates 49
9.2. ITU coordinates 49
9.2.1. ITU object-based polar coordinates 49
9.2.2. ITU object-based Cartesian coordinates 49
9.2.3. ITU scene-based polar coordinates 49
9.2.4. ITU scene-based cartesian coordinates 50
9.3. Navigation coordinates 50
10. Libraries 50
10.1. ParamSet and Patch libraries 51
10.1.1. ParamSets 51

10.1.2. Patches 52
10.2. Program Libraries 52
10.3. Proprietary libraries 52

11. Tasks 52

12. Sessions 53

13. Security 53

14. Concurrency control 53

15. Reliability 54

15.1. General 54
15.2. Availability 55
15.2.1. Keep-alive messages 55
15.2.2. Efficient reinitialization 55
15.3. Robustness 55

16. Device reset 55

17. Firmware and software upgrade 56

17.1. Update Types 56
17.2. Update Modes 57
17.3. Mechanisms 57
17.3.1. Active Updating 57
17.3.2. Passive Updating 57

Annex A. (Informative) – Actuator Example 59

A.1. General 59
A.2. Properties, Methods, and Events 59

Annex B. (Informative) – Block examples 61

A. Simple microphone channel 61
B. Two-channel microphone mixer 61
C. Mixer using nested blocks 62

Annex C. (Informative) – Other Media Network Control Standards 63

A. SMPTE ST 2071 - Media Device Control 63
B. Architecture for Control Networks (ACN) 63
C. Open Sound Control (OSC) 63
D. Ember+ 63

Annex D. Bibliography 64

Foreword

This foreword is not part of this document, AES70, *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 X210 to be developed in task group SC-02-12-L. The OCA Alliance also contributed to the task-group working draft and, as a result, there are various 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

Foreword to the 2018 edition

The members of the writing group that drafted this version of the document are: F. Bergholtz, J. Berryman, A. Gödeke, J. Grant, A. Kuzub, M. Lave, G. Linis, S. Price, M. Renz, A. Rosen, S. Scott, G. Shay, P. Stevens, P. Treleaven, S. van Tienen, and E. Wetzell. As well, the writing group was materially assisted by contributions from T. de Brouwer, B. Escalante, S. Jones, M. Smaak, and M. Versteeg.

J. Berryman led the writing group.

Morten Lave
Chair, working group SC-02-12
2018-12-20

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".

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 (AES70) 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 controller which is based on a newer version 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 controller which is based on an older version 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)

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 a term is first introduced in body text, the term will be set in an *italic* typeface.

When normative references are cited in the text they are [enclosed in brackets].

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 AES70-2: Class Structure, and AES70-3: OCP.1 Protocol for IP Networks.

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.

AES17. *AES standard method for digital audio engineering - Measurement of digital audio equipment*, Audio Engineering Society, New York, NY., US.

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.

IEEE-1588. *1588-2008 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, Institute of Electrical and Electronic Engineers (IEEE), Piscataway, New Jersey, US.

ISO-9787. *ISO 9787:2013 - Robots and robotic devices -- Coordinate systems and motion nomenclatures.* International Standards Organization (ISO), Geneva, Switzerland.

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.

ITU-1(8). *ITU-R BS.2076.1, Audio Definition Model*; chapter 8, *Coordinate System*. International Telecommunications Union, Geneva, Switzerland, 2017 June.

SMPTE-2059-2. *SMPTE Profile for Use of IEEE-1588 Precision Time Protocol in Professional Broadcast Applications.* Society of Motion Picture and Television Engineers, Piscataway, NJ, US.

WGS-84. US Department of Defense World Geodetic System 1984. Third edition, NIMA TR8350.2. National Imagery and Mapping Agency, Washington, DC, 2000 January 3.
http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf.

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 AES70 control object. Object Numbers are unique within each given device.

3.15. Precision Time Protocol (PTP)
the time protocol defined by [IEEE-1588].

3.16. Program or AES70 Program
an execution specification that is run within a device by an AES70 Task

3.17. Task
AES70 Task
a process that runs an AES70 Program within a device

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

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

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

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

3.22. Alignment level
a defined anchor point that represents a reasonable or typical level. See [Wiki-001].

4. Top level design

4.1. General

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

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. General

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.

   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. Content

Every AES70 class shall consist of the following:

1. A set of elements (properties, methods, and events - see clause 4.2.2.7)
2. A unique *class identifier*.
3. Exactly one parent class (except for the root class, which has no parent).

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

4.2.2.2. 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.

4.2.2.3. Nonstandard 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 nonstandard class may inherit from any standard class, or from another nonstandard class.
2. A nonstandard class should be attached to the standard class tree at the most specific level that is appropriate.
3. Identifiers of nonstandard classes shall be appropriately constructed using *authority keys* - see clause 4.2.2.6, and clause 4.2.2.6.2.
4. Nonstandard classes shall be defined in accordance with the inheritance rules listed in clause 4.2.2.8.

4.2.2.4. Containment

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

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

4.2.2.5. Collection

In this specification, a class will sometimes be described as *collecting* another class. This means that 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.

4.2.2.6. Class identifiers

A class identifier (*class ID*) shall consist of a *lineage key* and a *version number*. Every class ID includes a version number to uniquely identify the revision level of the class.

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

4.2.2.6.1. Lineage keys

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

Any class index other than the first ($\boldsymbol{i_1}$) may optionally be preceded by an *authority identifier*. An authority identifier is a special binary construct that identifies the defining authority of the index values following. If no authority identifier appears in a lineage key, the identified class is a standard one whose authority is this AES70 standard. Authority identifiers are discussed further in 4.2.2.6.4 below.

The lineage key of each class shall consist of a set of class indices, possibly prefixed by authority identifiers, 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 standard 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**.

4.2.2.6.2. Standard class IDs

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.

4.2.2.6.3. Nonstandard class IDs

Nonstandard class IDs may be created by the manufacturer of a device or by any other organization that develops class definitions. The source of a class ID is called the class's *authority.*

A nonstandard class ID shall be constructed by prefixing the nonstandard portion of the lineage key with an authority identifier. The authority identifier shall be inserted in the lineage key immediately before the index of the first nonstandard class. In the following details, authority keys will be indicated by "***A***", as in {$\boldsymbol{n};\ \boldsymbol{i_1{\bullet}i_2{\bullet}A{\bullet}i_3}$ ...**}.**

The interpretation of a lineage key that includes an authority identifier A shall be as follows:

1. Every index to the left of A shall identify a standard AES70 class;
2. Every index to the right of A shall identify a nonstandard class defined by the authority identified by an authority key (see clause 4.2.2.6.5); and
3. The index immediately to the left of **A** shall identify a class is the parent of the class whose index is immediately to the right of **A**.

For example, the lineage key of {**1•1•2•A•5}** identifies a nonstandard class whose parent is {**1•1•2}**, whose authority is **A**, and whose own (nonstandard) class index is **5**.

All class indices to the right of the authority index shall be considered to be nonstandard. For example, the lineage key {**1•1•2•A•5•4}** identifies a nonstandard class with index **4** that is a child of the nonstandard class with index **5** that is a child of the standard class {**1•1•2}.**

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

4.2.2.6.4. Lineage key format

The normative specification of the lineage key format is given here.

1. Each index value $\boldsymbol{i}_k$ shall be a 16-bit unsigned integer.
2. To form the lineage key, index values shall be concatenated left to right in order of inheritance.
3. An authority key (see clause 4.2.2.6.5) shall be 48 bits long and may be inserted before any index value except the leftmost (root).
4. The leftmost index value shall be the index of the OCA root class, which always shall have the value 1.

Class index values may be freely assigned subject to the following rules:

1. The value 0 shall not be used.
2. Values 1...32767 (0001_{16}...$7FFF_{16}$) may be used for standard or nonstandard class indices
3. Values 32768...65279 (8000_{16}...$FEFF_{16}$) shall be reserved for legacy AES70-2015 nonstandard classes.
4. Values 65280...65534 ($FF00_{16}$...$FFFE_{16}$) shall be volatile and intended for testing.
5. Value 65535 ($FFFF_{16}$) shall be a special flag identifying the start of an authority identifier.

The use of values in the range 32768...65534 is **deprecated**.

4.2.2.6.5. Authority key

The format of an authority key shall be as follows (left to right, hexadecimal notation):

Bits 0-15	$FFFF_{16}$
Bits 16-23	00_{16}
Bits 24-48	Authority's IEEE Public Company ID (Public CID) or Organizational Unique Identifier (OUI)

NOTE: The IEEE (Institute of Electrical and Electronic Engineers) Public CID is a 24-bit identifier which uniquely identifies an organization. Any incorporated organization may receive a unique Public CID from the IEEE upon request and payment of a one-time fee. See IEEE-1 and IEEE-2. CIDs and OUIs are non-overlapping, so either will serve to identify the authority.

4.2.2.7. 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 clause 6.

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].

4.2.2.7.1. 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.

4.2.2.7.2. Method status

All methods shall return a status code which indicates successful completion or, if unsuccessful, the general reason for failure. Status code values are described by the **OcaStatus** datatype defined normatively in [AES70-2].

4.2.2.7.3. 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.

4.2.2.7.4. Text format

All text in AES70 properties, method parameters, and event parameters shall be in UTF-8 format. See [ISO/IEC 10646-1].

4.2.2.8. 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 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:
   - Adding new properties, methods, and/or events; and/or
   - 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. Basic mechanism

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 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 16), 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 6 , and shall be used by the device reset mechanism, described in clause 16.

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 clause 5.4.3 and *AES70-2*. Creation and deletion of objects are discussed further in clause 5.10.

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 may be re-used in any way that does not violate rules (1)-(5) below.

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 (clause 5.7) 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.

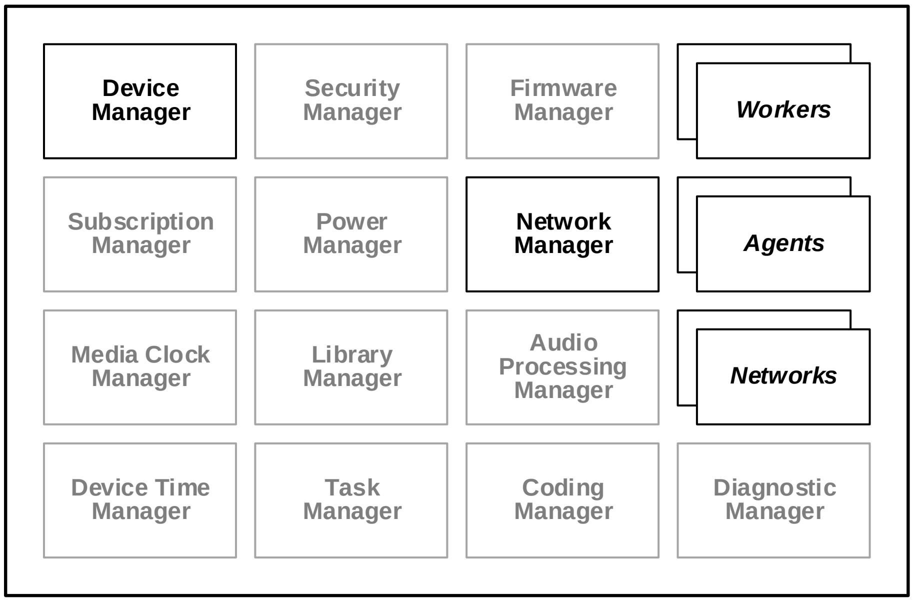


Figure 1 - AES70 device model. Optional managers are shown in grey.

5.3.2. Managers, Workers, and Agents

The AES70 device model contains four 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 clause 5.7 and [AES70-2].

2. **Workers**. Worker objects shall directly control the application functions of the device - see clause 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

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 clause 5.5.
4. **Networks**. Network objects describe external communication networks to which the device is connected. An external communication network may support media transport, control and monitoring, or both.

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 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 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 **Error! Reference source not found.** 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 clause 5.5.5 and clause 6.4).

5.4.3. Blocks

5.4.3.1. Basic mechanism

A block shall be a special kind of Worker that may contain Worker objects (including other block objects), Network 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, every Network, 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 the device's Workers, Networks, and Agents. In simple devices, all the Workers, Networks, and Agents may belong to the root block. In more complex devices, Workers, Networks, 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. It is important to remember that AES70 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.

5.4.3.2. Block enumeration

The set of Workers, Networks, 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 the objects it contains.

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 AES70.

5.4.3.5. Block searching

Every block shall include methods that allow controllers to search for objects the block contains. The following forms of search shall be supported:

1. Search for object(s) given a Role value; search either in the given block only, or in the given block and all contained blocks.
2. Search for object(s) given a Label value; ; search either in the given block only, or in the given block and all contained blocks.
3. Retrieve object(s) in the given block or any contained block, given a Path value. "Path" is defined in clause 5.9.

5.4.3.6. 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 clause 5.5.2.

5.4.3.7. Signal flow

5.4.3.7.1. Ports

For the purpose of defining signal flow, every Worker and every Network object may contain one or more *ports*. A port is a data element defined by the **`OcaPort`** class 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".

5.4.3.7.2. 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.

5.4.3.7.3. 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.

5.4.3.7.4. 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 clause 7.

`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.

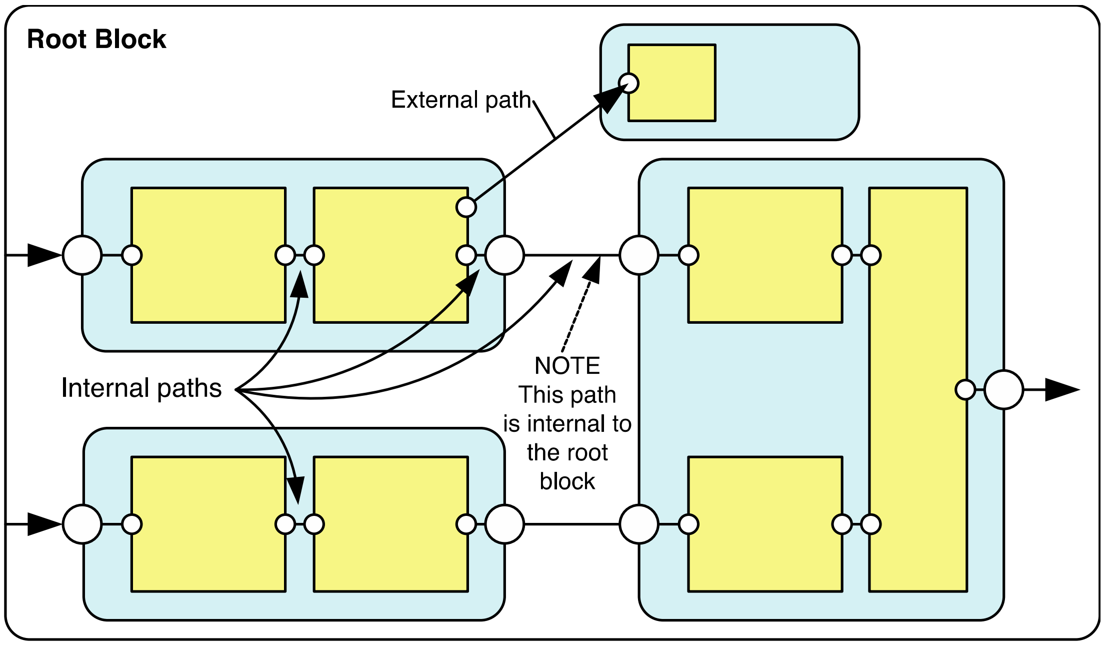


Figure 2 - Signal flow

NOTE. Illustrations 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.

5.4.3.8. Examples

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

5.4.3.9. 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 an 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. When each prototype object is defined, its author shall assign it a prototype object number. When the block factory constructs a new block, the prototype object shall be replaced in the new block by actual object numbers. The ONoMap property of the new block shall contain a map that specifies the correspondence between prototype object numbers and the actual object numbers in the new block.

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

a. 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.

b. 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.

5.4.3.10. 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.

5.4.3.11. Reusable blocks

The AES70 Reusable Block feature shall provide a mechanism whereby well-known block definitions can be maintained outside of particular device implementations and be reused in multiple devices. The reusable block feature shall be part of the AES70 standard, but the definitions of specific reusable blocks shall not.

NOTE: The rationale for the reusable block mechanism is to:

- Allow manufacturers to define standard blocks for all their products.
- Allow standards organizations to define standard blocks for general industry use.
- Allow the creation of AES70 profiles for categories of devices or device elements without having to include such profiles in the standard.

5.4.3.11.1. BlockType ID

A reusable block shall be constructed in the same way as a non-reusable block except that the reusable block's **`GlobalType`** property shall be set to a unique BlockType ID - see below. In a non-reusable block, the **`GlobalType`** property shall be zero.

> NOTE: The purpose of the BlockTypeID property is to give controllers a way of recognizing reusable blocks.

A BlockType ID shall consist of a unique authority identifier that identifies the owner of the reusable block's definition, followed by a sequence number that is unique within the AES70 environment of that owner. Authority identifiers shall be IEEE OUI or CCI values, the same as are used in the Class Identifier - see clause 4.2.2.6.4.

The normative definition of a BlockType ID is given in the **`OcaBlockType`** datatype defined in AES70-2.

Reusable blocks shall be instantiated in a device in the normal ways for Blocks, i.e. either built in at time of manufacture, or created dynamically at a later time.

5.4.3.11.2. Reusable-block factories

A *Reusable-Block Factory* is an OCA Block Factory that defines a reusable block. In some cases, this factory may construct that reusable block dynamically; in other cases, the factory's definition may simply serve as a machine-readable document that describes the reusable block's contents.

A reusable-block factory shall be the same as a non-reusable-block factory except that the reusable block's BlockType property shall be set to a unique BlockType. In a non-reusable block, the BlockType property shall be zero.

5.4.3.11.3. Object numbers in reusable blocks

When a reusable block is instantiated, the objects it contains shall be given new, unique object numbers. To make it easy for a controller to discover these object numbers, each **`OcaBlock`** shall contain an object number map, or **`ONoMap`**. The **`ONoMap`** shall be an **`OcaMap`** property that contains one entry for each prototype object number in the defining factory, and shall specify the mapping of prototype object numbers to actual object numbers.

5.4.4. Matrices

5.4.4.1. General

The AES70 matrix is 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.

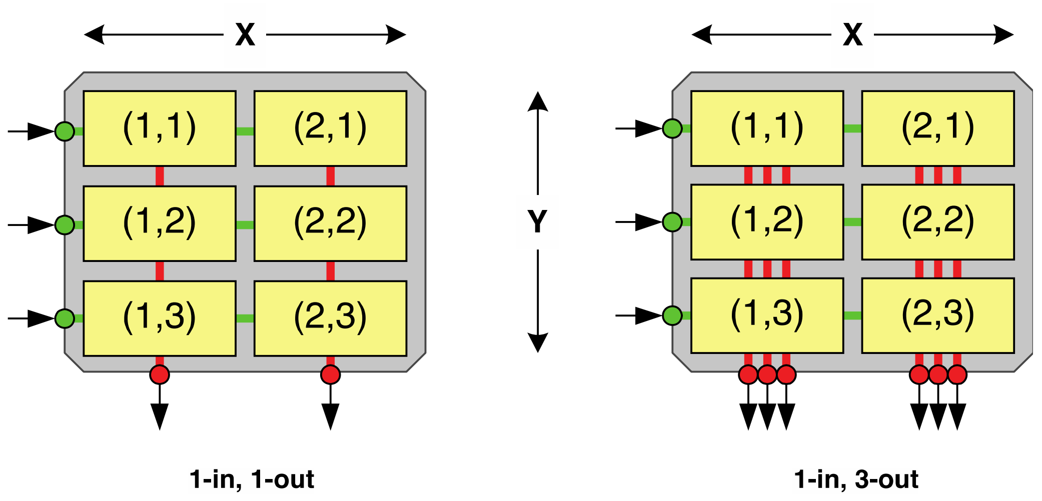


Figure 3. AES70 Matrices

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.

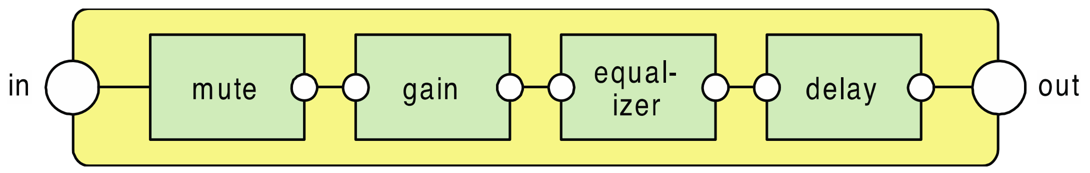


Figure 4. A complex matrix element

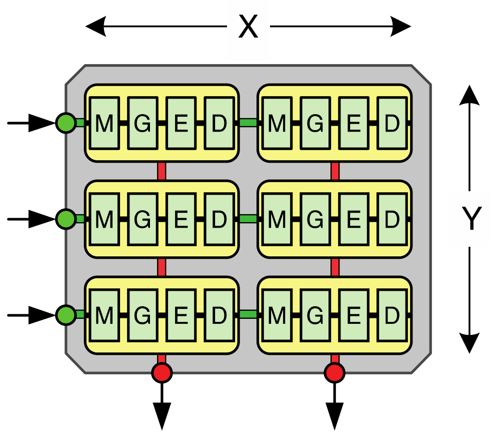


Figure 5. A matrix of complex elements

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

A matrix may contain other matrices, but it may not contain itself.

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 clause 5.1) of the device.

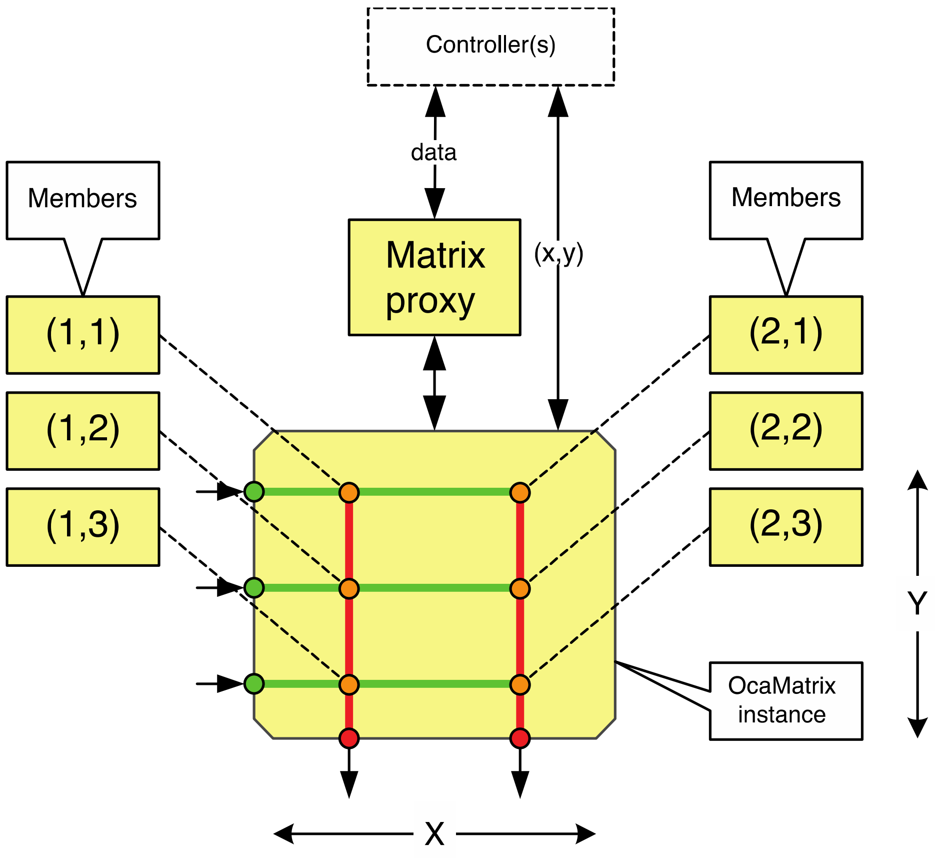


Figure 6. AES70 matrix structure

5.4.4.5.1. Matrix access behavior

1. When a Set operation on a matrix proxy fails to set the value of the property in one or more matrix members, then no matrix member property value shall be set. In other words, a Set operation on a matrix proxy shall succeed in setting the value in all affected members or none of them, but partial results shall not be allowed.
2. When a matrix is locked, none of its members shall be destructed. Violation of this rule shall be a programming error. More information about locking is in clause 14.

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:

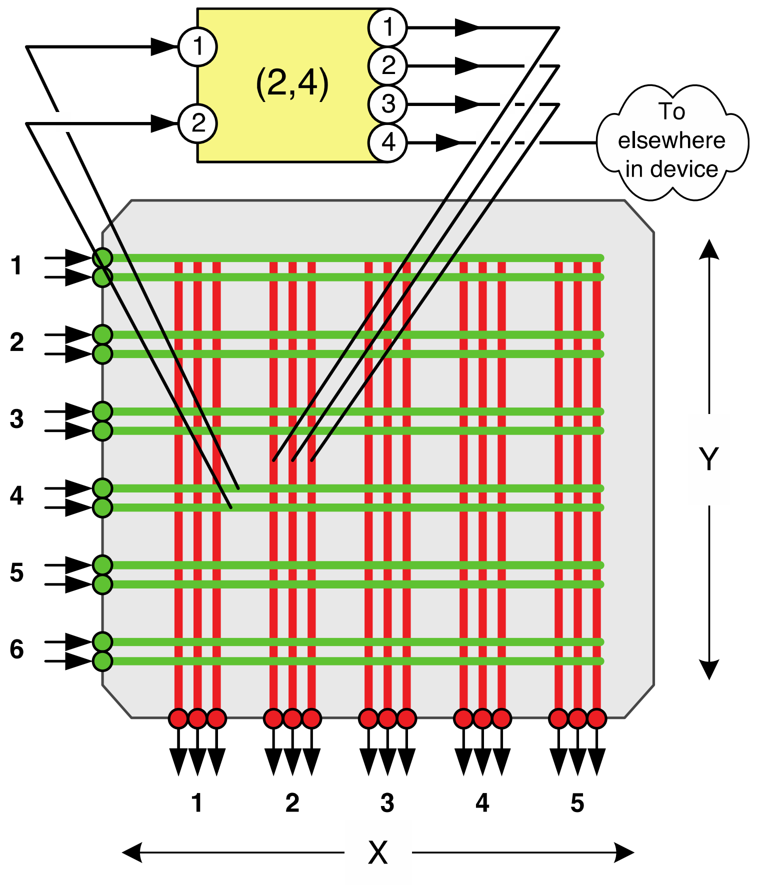
1. The ***i***th input port of a member in row ***y*** should be connected to matrix port $\boldsymbol{i + N_{in} (y-1)}$.

2. The ***j***th output port of a member in column ***x*** should be connected to matrix port $\boldsymbol{j + N_{out} (x-1)}$.

A matrix's members should have sufficient numbers of ports to support the above rules. Specifically, each member should have at least $\boldsymbol{N_{in}}$ input ports and $\boldsymbol{N_{out}}$ 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 5.5.2.

5.4.4.8. Application notes

5.4.4.8.1. Non-mixing applications

Matrices are defined rather generally and 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 would normally be summed to create the final column signal output. However, the AES70 control abstraction does not require such behavior; the matrix object may use 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. OcaGrouper

5.5.2.1. Basic mechanism

`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 may 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.

NOTE 2. A *sensor grouper* allows many sensor objects to be observed using a single output value. Sensor groupers will be defined in a subsequent version of this standard. This version of AES70 defines only actuator groupers.

5.5.2.2. Groups

A grouper shall contain one or more *groups.* A group shall be a collection of Workers or Agents. If all the objects in a grouper are of the same class, the grouper shall be termed a *simple grouper.* If not, the group shall be termed a *complex grouper.*

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 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 clause 5.5.2.4.1.

5.5.2.3. 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. 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. 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 for AES70.

5.5.2.4. Group structure

5.5.2.4.1. Basic mechanism

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 (see clause 5.5.2.4.2-clause 5.5.2.4.4 below)

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.

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.

5.5.2.4.2. 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.

5.5.2.4.3. 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.

5.5.2.4.4. Simple groups, complex groups, and group proxies

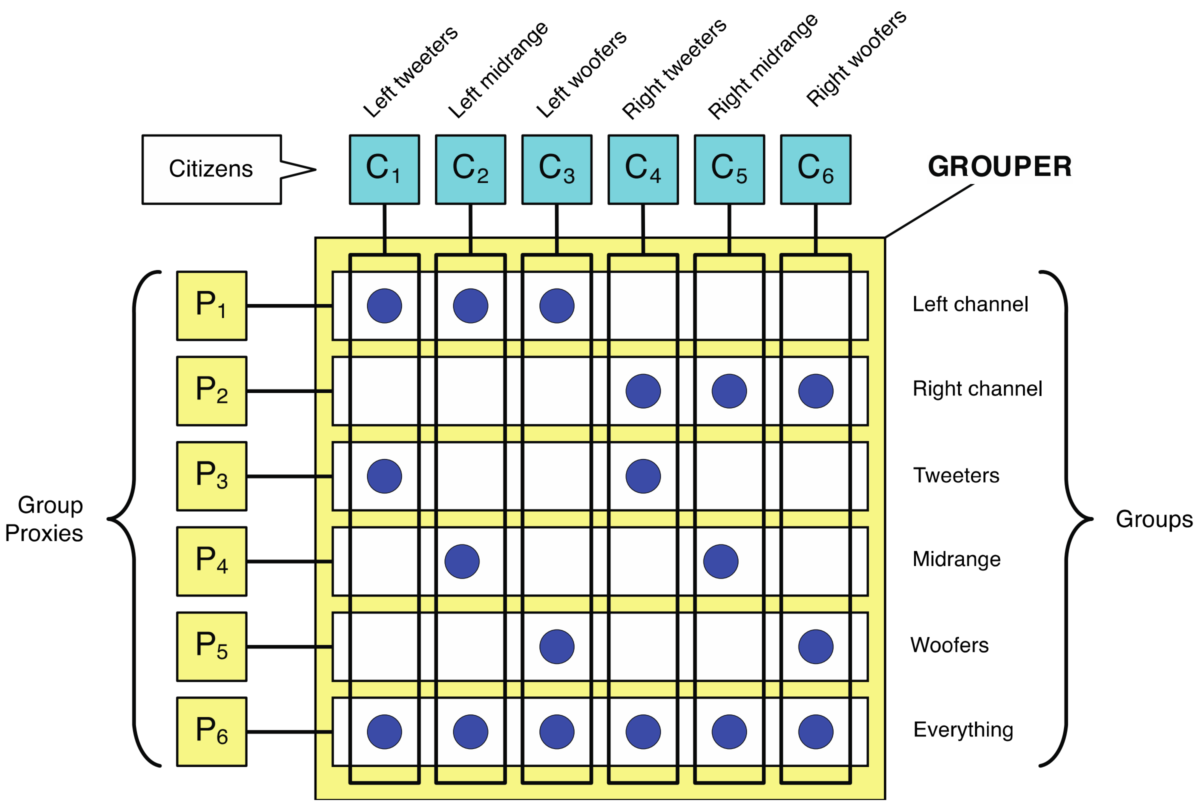
In a simple grouper, all the citizens are of the same class and group proxies, where used, shall be of this same class.

In a complex grouper, the citizens will, by definition, be of diverse classes; in this case, the group proxy class shall be the most specific class from which all the citizens inherit.

NOTE. For a complex grouper, only the properties defined in the group proxy are grouped.

5.5.2.4.5. Brief example

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



Squares indicate objects
circles indicate enrollments

Figure 8. Typical grouper

5.5.2.5. Aggregation and saturation rules

5.5.2.5.1. 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.

5.5.2.5.2. 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.

5.5.2.5.3. 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.

5.5.2.5.4. 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 to be informed of citizen connection losses.

5.5.2.5.5. 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 in cases where the lost citizen's setpoint values have not changed during the time it was unavailable.

Table 3 - Default aggregation rules

Base Datatype	**Aggregation Rule**
Float	SUM
Integer	SUM
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 groupers are deployed is a design choice of products and/or systems.

5.5.3. 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 AES70 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.

5.5.4. OcaTask

See clause 11.

When a device defines one or more **OcaTask** instances, the manager **OcaTaskManager** shall be instantiated to collect those instances.

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 clause 6.

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

See clause 9.

5.5.7. OcaMediaClock3

OcaMediaClock3 shall describe a particular internal or external media clock which the device uses. It shall include features for specifying available and current clock frequencies, and shall optionally link to an **OcaTimeSource** object that describes the time reference source the media clock uses.

A device may define any number of media clocks; each one shall be represented by its own instance of **OcaMediaClock3**. All 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 **OcaMediaClock3**.

5.5.8. OcaTimeSource

OcaTimeSource shall describe an internal or external time reference to which the device's media clocks may be synchronized.

An **`OcaTimeSource`** object may be the reference for any number of **`OcaMediaClock3`** objects, and a device may instantiate any number of **`OcaTimeSource`** objects. All instances of **`OcaTimeSource`** shall be collected by **`OcaMediaClockManager`**.

5.5.9. OcaPhysicalPosition

`OcaPhysicalPosition` shall describe the physical position of a real or abstract point in space. The position may be reported in various coordinate systems - see clause 9.

5.5.10. OcaEventHandler

See clause 6.2.

5.6. Network classes

See clause 7.

5.7. 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 clause 6.
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 libraries, the AES70 elements that handle pre-stored device configurations and parameter settings, predefined programs for tasks to run, and optional proprietary stored elements. See clause 9.
9	**`OcaAudioProcessingManager`**	Manages global audio processing parameters.
10	**`OcaDeviceTimeManager`**	Provides access to device time-of-day clock.
11	**`OcaTaskManager`**	Manages tasks. See clause 11.
12	**`OcaCodingManager`**	Manages the device's media encoding and decoding repertoire. See clause 7.3.2.6.

ONo	Class name	Function(s)
13	**`OcaDiagnosticManager`**	Provides functions to aid in network system setup and diagnosis
100	**`OcaBlock`**	Root block. See clause 5.4.3.

NOTE. Some manager objects shall be required for all devices, others are optional - see *AES70-2 Annex A*.

5.8. Standard Object Numbers (ONo)

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

5.9. 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".

The term *role path* or just *path* shall mean the ordered set of Role values of the hierarchy of blocks that contain an object, beginning with the Role value of the root object and ending with the Role value of the object itself.

5.10. Constructing objects

5.10.1. 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 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.

5.10.2. 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.

5.11. 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 shall be deletable.

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

When a Worker or a Network 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*, and the receiving object is termed the *subscriber*. 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.

Subscriptions shall be set up by the **`OcaSubscriptionManager`** object (see clause 5.7); a request to the **`OcaSubscriptionManager`** to set up a subscription is called a *subscription request.* Parameters in a subscription request are called *subscription parameters.*

Receipt of a notification shall generate a call to a specific method (the *subscribed method*) of the subscriber. The specific subscribed method is identified by a parameter in the subscription request.

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 clause 4.3.2.

When the Subscription Manager receives a request for a Fast subscription, but the current message delivery service does not support it, the Subscription Manager shall respond to the request with a Not Implemented status.

6.1.3. Lightweight subscriptions

A subscription mechanism shall contain two elements that shall be specified at the time the subscription is created, and whose values the device shall return to the controller in notifications: (a) the subscriber method identification, which is the identifier of the method in the controller to which the notification shall be directed; and (b) the *subscriber context*, an arbitrary value intended to assist the controller in processing the event.

Devices that do not have sufficient storage to recall these quantities may support *lightweight subscriptions*, for which:

1. The subscriber method identifier, which normally is the combination of an object number and a method ID, shall be entirely zeros; such a method ID shall be termed a *null method identifier;* and
2. The subscriber context is a zero-length bitstring.

Normative details of this mechanism are in AES70-2 Annex A, in the definition of the **`OcaSubscriptionManager`** class.

6.1.4. 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 clause 15.2.1.

When a device detects a subscriber failure, it shall delete all subscriptions which were made by the failed subscriber, except possibly for subscriptions which have been aggregated - see clause 6.1.5.

6.1.5. Subscription aggregation

For network and processing efficiency, devices may optionally perform *subscription aggregation*, which is a way of transmitting single notifications for multiple identical fast subscriptions that use a common multicast address.

Subscription aggregation shall function as follows:

1. When a device creates two or more Fast subscriptions for an emitter, and when the available Fast delivery service supports multicasting, and when the subscription parameters for those subscriptions obey certain conditions, then the device shall send each notification for those subscriptions via a single multicast message, rather than as individual messages to each subscriber. The set of such subscriptions is called a *subscription group.*
2. Subscriptions shall be aggregated when and only when the subscription parameters for all subscriptions involved shall:
   a. Specify Fast transmission;
   b. Specify the same (non-null) multicast address; and
   c. Specify the same subscriber method.

3. When an aggregated subscription is deleted, the subscription shall persist for the other members of its subscription group. When the last subscription of the group is deleted, the subscription shall be deleted from the device.

6.2. 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 1. Although AES70 places no limit on numbers of subscribers or subscriptions, in practice there will be implementation limits that vary from device to device.

NOTE 2. AES70 does not prohibit a given event handler from having multiple simultaneous subscriptions to a given event; however, it is recommended this practice be avoided.

If a notification is delivered to a device that has no subscription for that notification, the device shall ignore the notification without indicating a protocol error.

NOTE: The purpose of this rule to remove possible race conditions during subscription deletion.

6.3. 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 a device's controllable parameters reside in its properties, the **`PropertyChanged`** event gives complete access to the device's controllable operating state.

6.4. Use of numeric observers

The **`OcaNumericObserver`** class is defined in *AES70-2 Annex A*. 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.

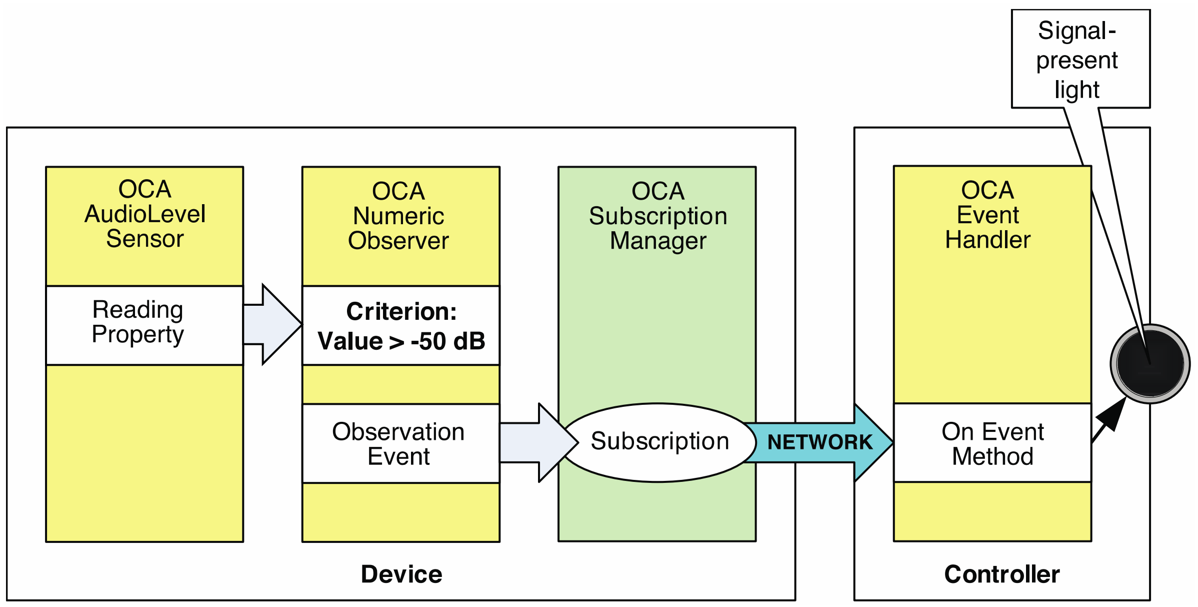


Figure 9. Using a numeric observer to implement a signal-present light

7. Networking and connection management

7.1. General

In AES70, the concept of network is generalized: A network may support control and monitoring, or streaming media transport, or both.

- For control and monitoring, AES70 defines functions for managing the exchange of control and monitoring data over external communications networks.

- For streaming media transport, AES70 defines functions for setting up, tearing down, and monitoring media transport connections over external communications networks.
- AES70 does not specify media transport services or define media transport architectures or protocols. The intent of AES70 is to co-operate with whatever media transport implementations implementers choose to use.

7.2. Connection Management Versions

The focus of AES70's networking scheme is on connection management, and especially media transport connection management. Considering this focus, AES70's networking schemes are identified as *AES70-CMn,* where (n) is a version number.

- AES70-2015's networking scheme is identified as *AES70-CM2*[1].
- The AES70-2018 scheme described in this document is identified as *AES70-CM3*, or, where the context is clear, simply *CM3*.
- For CM3, the networking classes have been considerably revised. As a result, a number of classes described in CM2 (i.e. in AES70-2015) are now deprecated. This section describes the CM3 connection management scheme. The deprecated classes of CM2 are defined in AES70-2 Annex A, in UML packages designated for deprecated definitions.

CM3-compliant controllers shall use **`OcaMediaTransportNetwork`** and its associated classes to access media network connection management functions for setting up, monitoring, and tearing down media transport connections between devices.

A device that implements CM2 shall be capable of making media connections to a device that implements CM3, and vice versa, as long as there is a CM2-compliant controller available to control the CM2 device and a CM3-compliant controller to control the CM3 device.

A device may implement both CM2 and CM3 connection management interfaces at the same time.

7.3. The CM3 object model

7.3.1. Classes

CM3 classes are listed below in Table 5. This table shows not only functional classes, but also key datatype classes.

Following Table 5, Figure 10 illustrates the key elements of the CM3 object model.

Further descriptions of the classes may be found beginning in clause 7.

Normative definitions of these classes are in AES70-2 Annex A.

1 AES70-CM1 was a developmental scheme that was never standardized.

Table 5. AES70-CM3 classes

Name	Type	Function(s)
`OcaNetworkManager`	Manager	Collects all network and codec classes
`OcaApplicationNetwork`	Network	Abstract base class from which **`OcaControlNetwork`** and **`OcaMediaTransportNetwork`** are derived
`OcaControlNetwork`	Network	Communication network for the transport of control and monitoring information
`OcaMediaTransportNetwork`	Network	Communication network for the transport of streaming media
`OcaMediaSourceConnector` **`OcaMediaSinkConnector`**	Datatype	Data structures that define media stream connection endpoints
`OcaMediaConnection`	Datatype	Data structure that defines a media stream connection
`OcaMediaStreamParameters`	Datatype	Data structure that specifies a media steam's properties
`OcaMediaCoding`	Datatype	Data structure that contains media encoding/decoding parameters

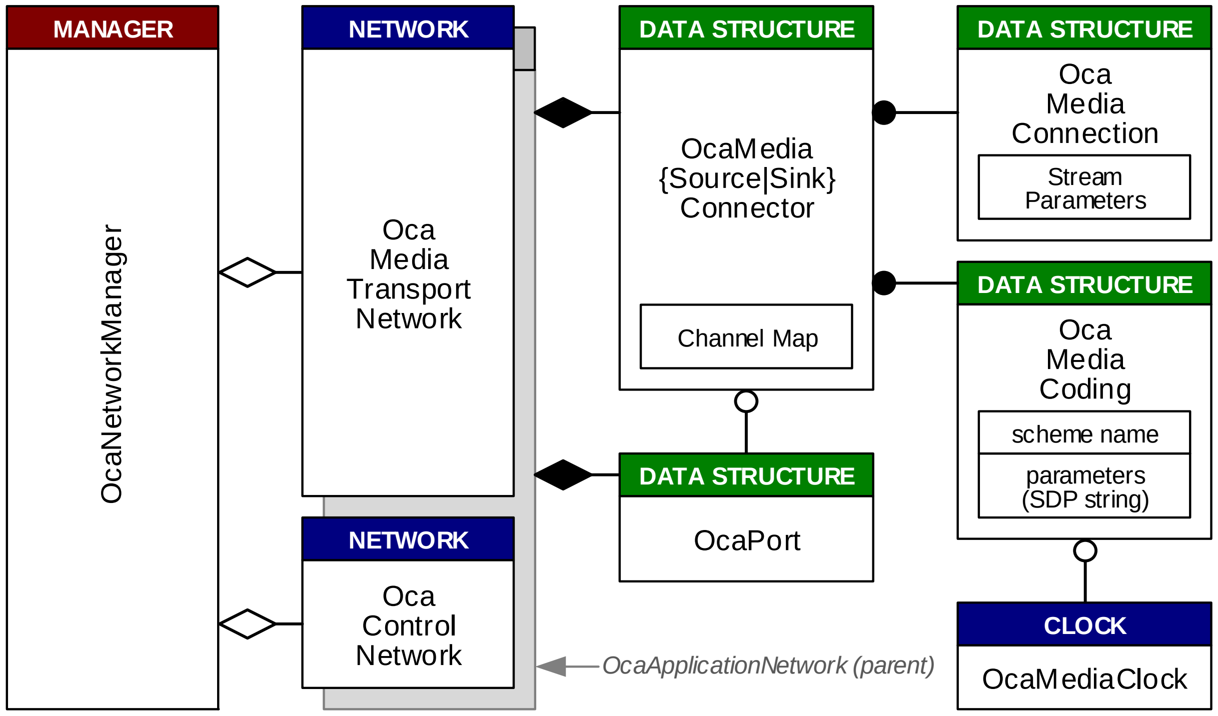


Figure 10. AES70-CM3 object model

NOTE: In Figure 10:

- A solid diamond- or circle-headed arrow represents a containment relationship, in which instances of the class at the plain end are contained within the class instance at the headed end.
- An open diamond- or circle-headed arrow represents a reference collection, in which the class instance at the headed end contains references to class instances at the plain end, but does not contain those instances.

- A diamond head indicates zero or more instances; a circle head indicates zero or one instance.

A CM3-compliant device shall implement at least one instance of **OcaControlNetwork** and zero or more **OcaMediaTransportNetwork** objects. Devices that implement no **OcaMediaTransportNetwork** objects are referred to as *control-only* devices.

When a device uses a single communication network for transport of both control information and streaming media, the device shall implement both an **OcaControlNetwork** object and an **OcaMediaTransportNetwork** object, and both these objects shall be bound to the one communications network.

If a device belongs to multiple communication networks, multiple instances of either or both the network class(es) shall be used as required.

AES70-2015 (i.e. CM2)-compliant devices may implement deprecated predecessors of these classes - these are defined normatively in AES70-2 Annex A.

7.3.2. Structure

CM3 shall be organized around the **OcaMediaTransportNetwork** class. Each **OcaMediaTransportNetwork** instance shall provide the following features for its network:

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 media transport protocol(s) the network uses.
7. Collections of associated classes (see below).

OcaMediaTransportNetwork objects shall hold two key collections - *media connectors* and *OCA ports*.

7.3.2.1. Media connectors

A media connector shall be a connection endpoint for media streams. A media connector may be for output (a "source" connector) or input (a "sink" connector).

A media connector is analogous to an analogue electronic multiconnector, in that it can carry multiple signals. CM3 uses the abstract term *pin* to denote each signal passing through a connector. For example, a media connector capable of supporting up to 64 signal channels would be said to have 64 pins.

CM3 datatype classes **OcaMediaSourceConnector** and **OcaMediaSinkConnector** shall describe output and input connectors, respectively. These classes shall describe data structures whose instances are collected in the **Connectors** property of **OcaMediaTransportNetwork**.

7.3.2.2. Media connections

A media source or sink connector shall be connected to zero or one media stream(s). A connection shall be described by an instance of the datatype class **OcaMediaConnection**. When a connector is connected to a stream, an instance of **OcaMediaConnection** shall be stored in the **Connection** field of the respective connector instance.

7.3.2.3. Media Streams

Each connected media stream shall be described by an instance of the datatype class **OcaMediaStreamParameters**. This instance shall be stored in the **StreamParameters** field of **OcaMediaConnection**.

7.3.2.4. OCA ports

Each **OcaMediaTransportNetwork** object shall contain a collection of OCA ports. An OCA port is an abstraction that represents a connection endpoint for an internal signal flow. Each OCA port shall be described by an instance of the datatype class **OcaPort**. See clause 5.4.3.7.1.

For each media transport network, the collection of **OcaPort** instances shall reside in the **Ports** property of **OcaMediaTransportNetwork**. Each such OCA port shall allow a single signal ("channel") of a network stream to be connected to signal flows inside the device.

7.3.2.5. Channel pin map

For each **OcaMediaTransportNetwork** object, the mapping between stream signals and OCA ports shall be flexible, and shall be defined in the **ChannelPinMap** properties of **OcaMediaSourceConnector** and **OcaMediaSinkConnector**. The mappings in these properties shall identify the OCA port(s) to and from which connector pin signals are routed.

Routing rules shall differ slightly for source and sink connectors, as follows for source connectors, each pin shall be routed to one OCA port at most; for sink connectors, each pin may be routed to multiple OCA ports. Thus, a stream output signal must arise from only one internal signal flow, whereas a stream input signal may be dispatched to any number of internal signal flows.

7.3.2.6. Media encoding and decoding

AES70 shall place no restrictions on media stream formats, sampling rates, encodings, or other media data representations. CM3 shall define a flexible scheme for managing connections of streams in any format(s).

In a CM3 device, a coding function shall be identified by an instance of the datatype **OcaMediaCoding**. This datatype shall contain:

1. the name of the coding scheme used;
2. other coding parameters, if any, and
3. the object number of the relevant **OcaMediaClock3** object, if used.

Each **OcaMediaConnection** data structure shall contain an instance of **OcaMediaCoding**. Thus, CM3 shall allow each media connector to specify its own kind of coding.

7.3.2.7. Alignment Level

Different devices may have different digital alignment levels (see clause 3.22). When media streams flow between devices of differing alignment levels, a gain or loss in apparent signal level results. To allow controllers to manage such differences, CM3 shall provide parameters that allow controllers to learn the digital alignment levels of devices being connected, and to adjust gain to compensate for those differences.

In particular, each instance of the **`OcaMediaConnector`** datatype shall include

1. a property **`AlignmentLevel`** that specifies the digital alignment level for signals flowing through the given **`OcaMediaConnector`** instance.
2. a property **`AlignmentGain`** that controllers may set to introduce signal gain (or loss) for all signals flowing through the given **`OcaMediaConnector`** instance.

Alignment Level values shall be given in *dBFS.* See [AES17]. For the purpose of defining Alignment Level, full-scale shall be as follows:

- For fixed-point signal codings, full-scale amplitude references the **`OcaMediaConnector`**-to-network interface.
- For floating-point signal codings, full-scale amplitude shall be the floating-point value 1.0.

NOTE 1: The full-scale value of a device's network interface may not necessarily be the same as the full-scale value of the device's internal signal paths. AES70 Alignment Level is concerned only with full-scale values of network interfaces. Internal digital scaling details are out of scope.

7.4. AES70 Adaptations

AES70 network classes 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 is termed an AES70 Adaptation. AES70 Adaptations contain rules for configuring and using AES70 networking classes, and may additionally define specific refinements (subclasses) of standard AES70 classes. AES70 Adaptations are outside the scope of this standard.

8. Time

8.1. Format

For absolute time values, AES70 object properties use the standard PTP format defined in [IEEE-1588]. This format is as follows:

- 48 bits: seconds since the PTP Epoch as defined in clause 8.2.
- 32 bits: nanoseconds since the most recent second.

For time offsets, AES70 object properties shall use a datatype consisting of a standard PTP value and a sign flag, to indicate whether the offset is positive or negative.

Normative definitions of these formats are in AES70-2 Annex A, in the UML package *Time Datatypes.*

8.2. PTP Epoch

AES70's time epoch shall be the PTP Epoch as defined in [SMPTE-2059-2], namely 1970-01-01 00:00:00 TAI, where TAI is international Atomic Time [Wiki-002].

8.3. UTC time and the NTP time format

Previous versions of AES70 used UTC time in the NTP time format for certain time-related properties. In the version defined here, those properties are deprecated. In this version, support of UTC time shall be optional. UTC time should not be used for new AES70-compliant products.

9. Physical Position Coordinate Systems

AES70 shall support various coordinate systems for setting and retrieving physical positions. The following rules and definitions apply to all of these coordinate systems:

- All distance values shall be in meters.
- All angle values shall be in degrees.
- *Situation* of a coordinate system means its position and orientation with respect to real-world coordinates. In some cases, situation is application-dependent and outside the scope of the standard. In other cases, the coordinate system shall have a standard definition.

Coordinate systems defined by AES70 are summarized in Table 6, and further described in the text following.

Table 6. Physical Position Coordinate Systems

Name	Description
`Robotic`	Six-axis position and rotation per [ISO-9787].
`ItuObjectBasedPolar`	Object-based audio per [ITU-1(8)]. Polar version.
`ItuObjectBasedCartesian`	Object-based audio per [ITU-1(8)]. Cartesian version.
`ItuSceneBasedPolar`	Scene-based audio per [ITU-1(8)]. Polar version.
`ItuSceneBasedCartesian`	Object-based audio per [ITU-1(8)]. Cartesian version.
`Navigation`	Standard earth navigation coordinates per [WGS-84].

Table 7. Position coordinate system attributes. See text below for definitions of symbols.

Name	Coordinates	See clause	Situation
`Robotic`	X, Y, Z, rX, rY, rZ	9.1	application-dependent
`ItuObjectBasedPolar`	ϕ, θ, R, Scale	9.2.1	application-dependent
`ItuObjectBasedCartesian`	X, Y, Z, Scale	9.2.2	application-dependent
`ItuSceneBasedPolar`	ϕ, θ, R, Scale	9.2.3	application-dependent
`ItuSceneBasedCartesian`	X, Y, Z, Scale	9.2.4	application-dependent
`Navigation`	Latitude, Longitude, Altitude	9.3	fixed to WGS-84 origin

9.1. Robotic coordinates

`Robotic` coordinates shall be ix-axis robotic coordinates of the form (X, Y, Z, rX, rY, rZ), where rX, rY, and rZ shall be rotation of the physical object around the given axis - X, Y, or Z.

When viewed from the positive ends of the respective axes, rX, rY, and rZ shall increase for anticlockwise rotation.

The X-axis shall point in the rZ=0° direction, and the Y-axis shall point in the rZ=+90° direction.

Situation shall be application-dependent, and therefore not specified by this standard.

Detailed definitions are specified in [ISO-9787].

9.2. ITU coordinates

ITU coordinates are for capturing, processing, and playback of immersive audio programming. There are four different ITU coordinate systems, as described next.

9.2.1. ITU object-based polar coordinates

`ItuObjectBasedPolar` coordinates shall be object-based polar coordinates for audio elements, as defined by [ITU-1(8)].

Coordinate values shall be of the form (ϕ, θ, R, Scale) where (ϕ, θ, R) shall define a point of a unit sphere, and Scale shall be the scale factor to scale the point to an absolute position, and

- ϕ shall be *azimuth*: - angle in the horizontal plane with ϕ = 0° pointing from the origin toward the front center position.
- θ shall be *elevation* - angle in the vertical plane with θ = 0° pointing from the origin toward the front center position.
- R shall be *radius*: distance of the point from the center of the unit sphere; thus $0 \leq R \leq 1$.

Detailed definitions are specified in [ITU-1(8)].

9.2.2. ITU object-based Cartesian coordinates

`ItuObjectBasedCartesian` coordinates shall be object-based Cartesian coordinates for audio elements, as defined by [ITU-1(8)].

Coordinate values shall be of the form (X, Y, Z, Scale) where (X, Y, Z) shall define a point of a unit cube, and Scale shall be the scale factor to scale the point to an absolute position.

Detailed definitions are specified in [ITU-1(8)].

9.2.3. ITU scene-based polar coordinates

`ItuSceneBasedPolar` coordinates shall be scene-based polar coordinates for audio elements, as defined by ITU-1(8)].

Coordinate values shall be of the form (ϕ, θ, R, Scale) where (ϕ, θ, R) shall define a point of a unit sphere, and Scale shall be the scale factor to scale the point to an absolute position, and

- ϕ shall be *azimuth* - angle in the horizontal plane with $\phi = 0°$ pointing from the origin toward the front center position.
- θ shall be *elevation* - angle in the vertical plane with $\theta = 0°$ pointing from the origin in the direction the listener (at the origin) considers to be vertical.
- R shall be *radius*: distance of the point from the center of the unit sphere; thus $0 \leq R \leq 1$.

Detailed definitions are specified in [ITU-1(8)].

9.2.4. ITU scene-based cartesian coordinates

`ItuSceneBasedCartesian` coordinates shall be scene-based Cartesian coordinates for audio elements, as defined by [ITU-1(8)].

Coordinate values shall be of the form (X, Y, Z, Scale) where (X, Y, Z) shall define a point of a unit cube, and Scale shall be the scale factor to scale the point to an absolute position.

Detailed definitions are specified in [ITU-1(8)].

9.3. Navigation coordinates

`Navigation` coordinates shall be standard world navigation coordinates, as used in conventional Earth maps and provided by satellite positioning system receivers and other navigation devices, and as standardized in [WGS-84].

Coordinate values shall be {Longitude, Latitude, Altitude}, as defined by [WGS-84].

Situation shall be as specified in the [WGS-84] coordinate system, which places the origin at the center of mass of the Earth.

10. Libraries

An AES70 *library* is a set of predefined data and/or program elements stored within a device.

An element stored in a library is called a *library volume*, or just *volume.* A volume may be of one of three standard types, namely:

1. *ParamSet*: a collection of stored control settings;
2. *Patch*: a collection of ParamSet assignments;
3. *Program*: an executable element for an OcaTask (see clause 11);

In addition to the three standard volume types, manufacturers may define additional proprietary volume types. Proprietary volume types are described in clause 10.3.

Subject to implementation limitations, a device may contain any number of libraries, and a library may store any number of volumes. However, any given library shall store volumes of only one type. The type of a library shall be the type of the volumes it stores.

Each library in a device shall be managed by its own **`OcaLibrary`** instance. All libraries shall be collected by the **`OcaLibraryManager`** object.

Libraries are an optional AES70 mechanism that need not be implemented for basic AES70 compliance. A device may implement any number of libraries of any assortment of types, or no libraries at all.

AES70 does not specify the binary format of a volume. The library mechanism shall treat all volumes as BLOBs. This standard includes primitives for upload, download, creation, and management of volumes, but not primitives for examining or manipulating volume contents.

10.1. ParamSet and Patch libraries

Volume types **`ParamSet`** and **`Patch`** shall be parameter storage elements whose general purpose is to store and restore device parameter settings values.

In what follows, **`ParamSet`** and Patch libraries are collectively referred to as "parameter libraries".

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

10.1.1. ParamSets

A **`ParamSet`** shall be a library volume that stores property values for an AES70 block (see clause 5.4.3). The block to which a **`ParamSet`** applies is called a *target block* of the **`ParamSet`**.

The act of installing a **`ParamSet`**'s values into a target block is termed *applying the **`ParamSet`** to the block.*

If a **`ParamSet`**'s target block is reusable (see clause 5.4.3.11), the **`ParamSet`** may be applied to any block of the reusable block's blocktype. In this case, the reusable block's blocktype is called the **`ParamSet`**'s *target blocktype.*

Read-only properties (for example, Class ID) shall not be stored by **`ParamSet`**s.

A **`ParamSet`** need not contain values for all the properties in its target block. Applying a **`ParamSet`** to a block shall change only the property values stored in that **`ParamSet`**, and shall leave other properties unchanged.

AES70 defines two ways of creating and storing **`ParamSet`**s in a 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 its property values as a **`ParamSet`** in a particular library in the 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.

10.1.1.1. ParamSet Assignments

A ***`ParamSet`** Assignment* is a specification that defines the application of a specific **`ParamSet`** to a specific block. Because a given **`ParamSet`** may be applied to multiple blocks of the same blocktype, a single **`ParamSet`** may be involved in multiple assignments.

> EXAMPLE: Suppose a mixing console defined a blocktype 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 assigned to any instance of **`InChannel`**.

10.1.2. Patches

A Patch shall be 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 shall affect only the parameter values included in that **`Patch`**'s **`ParamSets`**; 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.

10.2. Program Libraries

A *program library* shall store Programs. A Program shall be an executable programmatic action or sequence that may be run by an AES70 Task, under control of the **`OcaTaskManager`**. See clause 11.

The content of Programs is application-specific and outside the scope of AES70. AES70 does not define device programming or scripting mechanisms.

10.3. Proprietary libraries

A *proprietary Library* shall be a Library that stores Volumes of a proprietary type. The type of a Library and its volumes shall be specified by a Volume Type Identifier, normatively defined in AES70-2 Annex A, datatype class **`OcaLibVolType`**. The format of this identifier allows manufacturers to create their own volume type identifiers that are guaranteed not to conflict with those defined by other manufacturers or by this Standard.

> NOTE: Proprietary Libraries can be used for storing and retrieving any kind of product- or manufacturer- specific device information. In many cases, implementing the Proprietary Library mechanism can obviate the need for device FTP services.

11. Tasks

A Task is a process that runs inside a device. The executable content of a Task shall be a Program (see clause 10.2).

Creating, configuring, controlling, and deleting of Tasks shall be managed by the **`OcaTaskManager`** object. **`OcaTaskManager`** shall maintain a collection of Task descriptors to describe extant tasks. A Task descriptor is an instance of the **`OcaTask`** datatype.

Tasks are optional. They may be created at device initialization time, or dynamically by the use of the **`OcaTaskManager.AddTask()`** method. **`AddTask()`** and its counterpart **`DeleteTask()`** need be implemented only by devices that support dynamic creation and deletion of Tasks.

To run, a Task shall be assigned a Program, then started by the appropriate means. A Task may be started on demand by a controller command to **`OcaTaskManager`**, or automatically at a specified future time configured into the Task descriptor.

Normative definitions of these mechanisms are in AES70-2 Annex A.

12. 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 clause 5.5.2) - will aid such implementations.

13. 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 OCP.1 protocol (see AES70-3), 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.

An AES70 network may operate with a mix of secure and insecure devices.

NOTE 1: In principle, networks that mix secure and insecure devices are insecure.

NOTE 2. 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.

14. 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 a mutual exclusion ("mutex") mechanism designed to prevent some or all access to the object by anyone other than the lockholder. The AES70 lock mechanism shall support two levels of lock, as follows:

1. *total* locks, which prevent all access to the object by non-lockholders, and
2. *readonly* locks, which prevent non-lockholders from modifying object data, but allow them to retrieve it.

The owner of a lock (*lockholder*) shall be the AES70 controller that sets the lock.

The lock interface shall be defined in the **`OcaRoot`** class, and shall therefore shall 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 clause 4.3) in response to all locking-related commands.
3. The lock interface shall implement both total and readonly lock levels.
4. A locked object may be total-locked or readonly-locked, but not both.
5. A lockable object may be locked by at most one lockholder at a time.
6. A lockholder may upgrade its lock from readonly to total, or downgrade its lock from total to readonly; non-lockholders shall not be able to do this.
7. No lock shall survive a device reset.
8. Controller to device communication should be continuously monitored (see clause 15.2.1). When a controller's communication with a device fails for any reason, the device shall automatically remove all locks set by that controller.
9. 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.
10. In fully-configurable devices, locked objects shall not be deletable.

15. Reliability

15.1. 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.

15.2. Availability

15.2.1. 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 AES70 protocol specification. The nature of these messages will depend on the specific AES70 protocol being used.

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

15.2.2. Efficient reinitialization

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

15.3. 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 OCP.1 protocol (see AES70-3) runs on Ethernet, its implementation can take advantage of spanning-tree protocols to increase resistance to network link failures.

16. 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 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 specific 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.

17. Firmware and software upgrade

AES70 defines mechanisms for reliable upgrading of device firmware or software over the network. The implementation of such mechanisms is optional.

When AES70-compliant 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 13.

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.

17.1. Update Types

AES70 shall support devices with multiple firmware components per device, and shall allow devices to support any of the following four *update types:*

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 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 fails, 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 - in common language, *bricked.*

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 a 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 are liable to render devices bricked.

In what follows, the term *updater* shall refer to the AES70 controller that is driving the firmware update process.

17.2. Update Modes

AES70 supports 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.

Either update mode can support any of the four update types described clause 17.1. Choice of update type and mode is at the discretion of device designers.

17.3. Mechanisms

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

17.3.1. Active Updating

The active update of a device shall proceed as follows:

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.

   d. Call the **`OcaFirmwareManager.AddImage`**`()` method as many times as necessary to upload the complete component firmware image to the device.

   e. Call the **`OcaFirmwareManager.VerifyImage`**`()` method, which causes the device to verify the integrity of the uploaded component firmware image.

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

17.3.2. Passive Updating

The passive update of a device shall proceed as follows:

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.

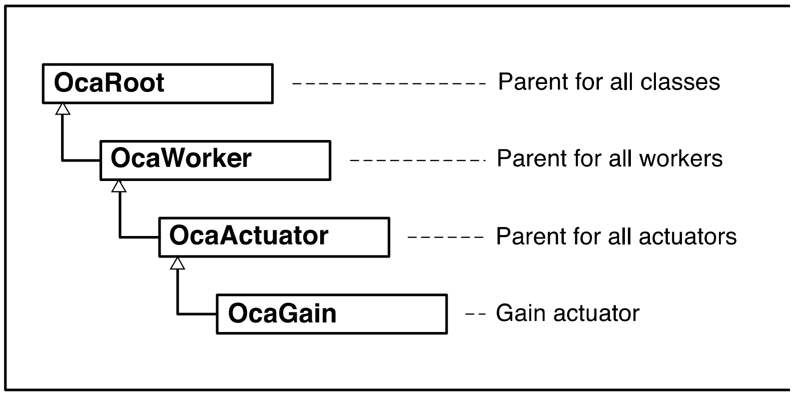


Figure A.1 - OcaGain lineage

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
ClassID	01p01	Class ID	RO	Class ID of OcaGain class	**OcaGain**
Class Version	01p02	Integer	RO	Version number of OcaGain class	**OcaGain**
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**
Gain	04p01	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

A. Simple microphone channel

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

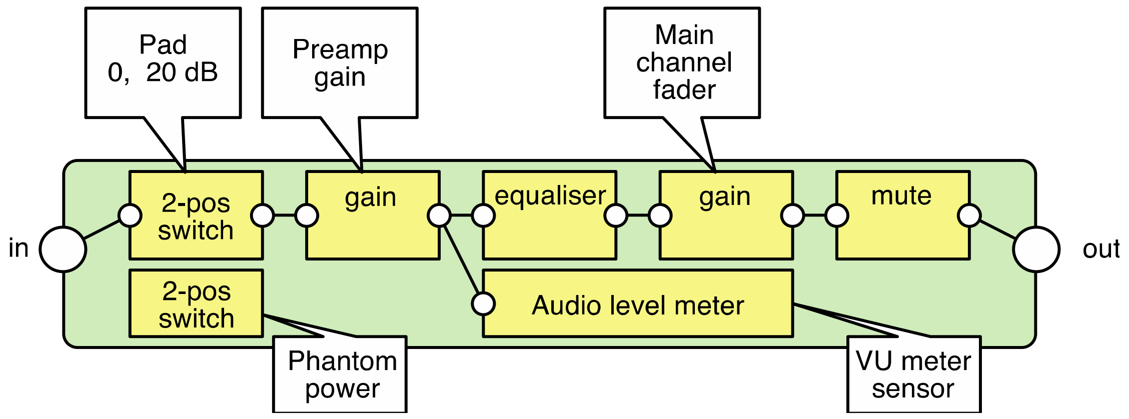


Figure B.1 - Simple microphone channel signal flow

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.

B. Two-channel microphone mixer

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

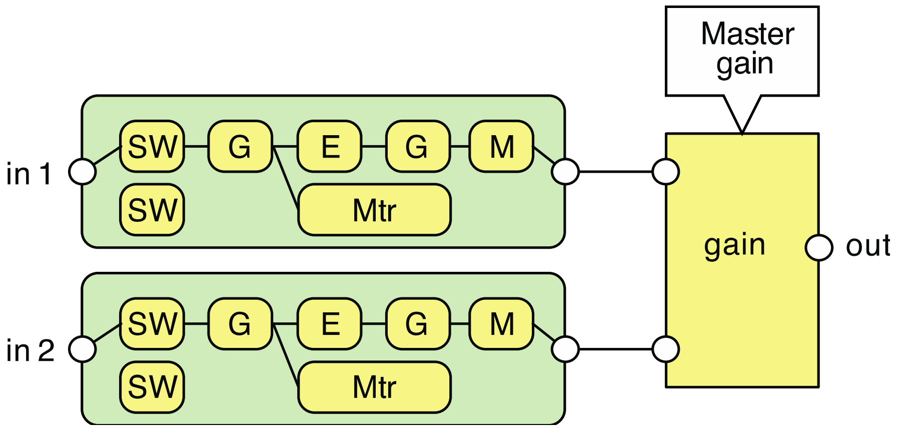


Figure B.2 - Two-channel microphone mixer

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

The reader will 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 AES70 signal flow might include an explicit summing point.

C. 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.

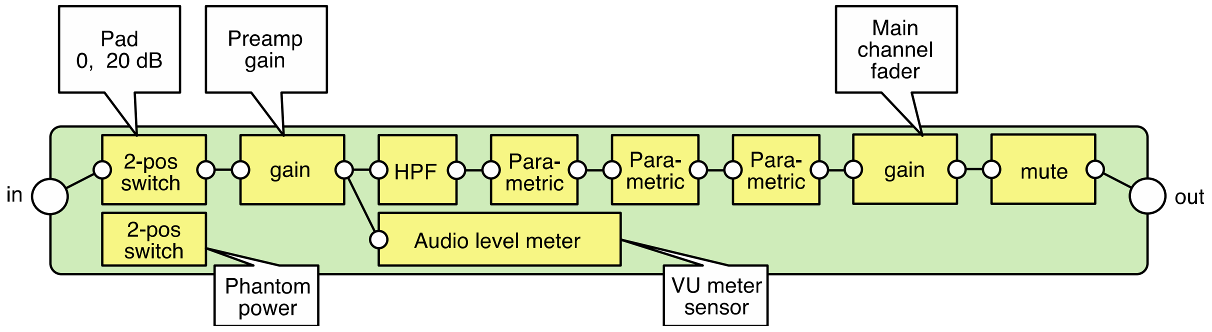


Figure B.3 - Mic channel with EQ sections inline

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. 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.

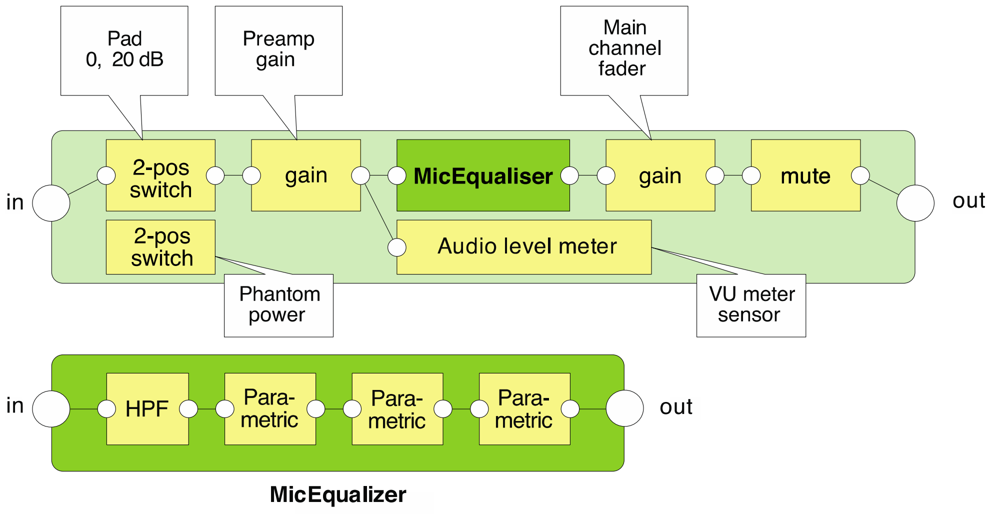


Figure B.4 - Microphone channel with MicEqualizer block

Annex C. (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.

A. 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/.

B. 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" . *Definition of ACN. Package of 17 documents plus supporting files.* Online:http://webstore.ansi.org

C. 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. 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 via GitHub, at https://github.com/Lawo/ember-plus/wiki .

Annex D. Bibliography

IEEE-1. *Guidelines for Use [of] Organizationally Unique Identifier (OUI) and Company ID (CID).* Institute of Electrical and Electronic Engineers. At https://standards.ieee.org/develop/regauth/tut/eui.pdf.

IEEE-2. *IEEE Registration Authority Home Page*. Institute of Electrical and Electronic Engineers. At http://standards.ieee.org/develop/regauth/index.html.

Wiki-001. *Alignment Level*. Wikipedia, https://en.wikipedia.org/wiki/Alignment_level.

Wiki-002. *International Atomic Time*. Wikipedia, https://en.wikipedia.org/wiki/International_Atomic_Time