OCA
Open Control Architecture

Release 1.2

OCF : Framework

Table of Contents

List of Figures

1. Introduction

This document describes the models and mechanisms of the Open Control Architecture, or OCA. These models and mechanisms are referred to as the OCA Framework, sometimes abbreviated as OCF.

OCA is a system control and monitoring architecture for media networks. OCA is for system control and monitoring only, and may be integrated with any streaming program transport protocol scheme, as long as the underlying digital network offers a suitable form of connectivity for OCA control and monitoring traffic.

Note: In what follows, the term control should generally be interpreted to mean control and monitoring.

1.1. Divisions of OCA

The divisions of OCA are:

1. OCF, the OCA Framework. Described in this document. OCF includes:
   - A generic communication model that can be used for various kinds of real networks, notably including TCP/IP Ethernets.;
   - A generic device model that defines the control interfaces that media devices present to the network;
2. OCC, the class structure which describes OCA's control repertoire in terms of an object-oriented model.
3. OCP.1, OCP.2, ... , the set of application protocols that implement OCA command and response sequences in real networks. There is one protocol for each kind of network that OCA supports, and each protocol is complete unto itself -- i.e., they are independent of each other. OCP.1 is the OCA protocol for TCP/IP Ethernet networks. OCP.2, OCP.3, etc. are to be defined.

   In what follows, the text will use OCP as a generic reference to OCP.1, OCP.2, and so on, with the understanding that the specific protocol chosen will depend on network type at hand.

1.2. References

In some cases, references denote a family of documents. Detailed documentation structure is defined in [OCADOC 1.1].

[OCADOC 1.1]	Guide to OCA documentation.
[OCC 1.1]	OCC - Class structure definition
[OCP.1 1.1]	OCP.1 - Protocol for TCP/IP Ethernet networks
[OCC-MIN 1.1]	Minimum object implementation for OCA-compliant devices

2. Architectural Goals and Constraints

OCA is based upon the following features and requirements:

2.1. Functionality

Provide the following functions:

- Discover devices

  Discover the OCA -compliant devices that are connected to the network.

- Manage media streaming connections

  Define and undefine media stream paths between/among devices. Interface with features of the media transport system to set up and take down media connections.

- Control devices

  Control operating and configuration parameters of an OCA -compliant device.

- Monitor devices

  Monitor operating and configuration parameters of an OCA -compliant device.

- Configure devices

  For devices with reconfigurable signal processing and/or control capabilities, define and manage configuration parameters.

- Upgrade software/firmware

  Upgrade software/firmware of controlled devices. Include features for fail-safe upgrades.

2.2. Security

Offer the possibility to enable the following security measures for control data:

- Entity authentication
- Prevention of eavesdropping
- Integrity protection
- Freshness

2.3. Scalability

- Physical distribution

  Impose minimal restriction on the physical distribution of application devices.

- Number of devices

  Control and monitor up to 10,000 (application) devices in one system.

2.4. Availability

Guarantee high availability by offering:

- Device supervision of OCA compliant devices.
- Link supervision of network links.
- Efficient network re-initialization following errors and configuration changes.

2.5. Robustness

Guarantee robustness by offering:

- A mechanism for operation confirmation
- A mechanism for handling loss of control data
- A mechanism for handling device failure of OCA compliant devices
- Recommendations on network robustness mechanisms that can be used

2.6. Safety Compliance

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

2.7. Compatibility

Detect different OCA network interface versions, and handle all of these versions. If incompatibilities do arise over versions, OCA must at least report which device is incompatible with the application. The design goal is an extensible protocol providing upwards and downwards compatibility.

2.8. Analyzability

Offer diagnostic functions to the application that allow access to the following information:

- Version information of all components (HW/SW) of a device
- Network parameters of a device
- Device status (including network status)
- Media stream parameters (for each active receive and/or transmit media stream of a device)
- Packet data errors

3. Top Level Design

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

OCA is intended to yield practical 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:

- OCA networks operate over industry-standard data network equipment.
- OCA networks can operate in a secure or unsecure mode, as the product range requires.
- OCA devices coexist harmlessly with non-OCA devices.
- OCA protocols are extensible; OCA allows the orderly incorporation of new device types and device upgrades, and generally supports upward compatible evolution of function. OCA also provides a well-defined mechanism for the incorporation of nonstandard devices in a maximally compatible way.
- OCA supports multiple protocol versions for different kinds of interconnection. The current version of OCA defines OCP.1, the protocol for TCP/IP Ethernets. Future versions will define OCP.2, OCP.3, etc., for other kinds of connections such as USB links.

3.1. Object Orientation

3.1.1. Concept

The OCA specification is expressed in object-oriented design terms. Communicating devices are described as sets of objects. Each object is an instantiation of a specific class, and has a state (i.e. a set of data elements) and a set of procedural actions ("methods") defined by that class.

All actions and features of OCA protocols are defined in terms of classes; the functional repertoire 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 called an "object-oriented protocol", and is completely defined by the union of the following four sets:

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

3.1.2. Classes

OCA classes are defined as nodes in a tree structure rooted in a single, elemental node (the “root class”), and arranged in order by inheritance. Inheritance means that each class is considered to be a specialized entity derived from another, less specialized class (the “parent class”). A class exhibits ("inherits") all the features of its parent, except where that class's definition specifically overrides an inherited feature.

Every OCA class consists of the following:

a. A set of elements (properties, methods, and events - see 3.1.2.2)

b. A unique class identifier.

c. Exactly one parent class, except for the base class, which has no parent.

In OCA, the rules for inheritance are:

a. A class implements all the properties, methods, and events of its parent class.

b. A class may override (i.e. redefine) one or more of its parent class’s methods

c. A class may define methods of its own that are not in the parent class.

d. Any given class except the base class inherits from exactly one other class.

These particular rules guarantee that new features can be added to the protocol in an upwards-compatible manner, by creating new classes from the old ones. Inheritance ensures that the new classes will support, at a minimum, the functions and interfaces of their parents.

Standard class names all begin with "Oca".

3.1.2.1. Class IDs

Each class is identified by a hierarchical ID of the form i.j.k. ... where i, j, k,... are integers that uniquely identify the class within its siblings in the class tree. Thus

- 1 designates the root class.
- 1.1 designates the first child of the root class.
- 1.2 designates the second child of the root class.
- 1.2.4 designates the fourth child of the second child of the root class
- et cetera

The actual format has a few more features. The datatype **OcaClassID** is defined as a variable-length vector with the following elements:

n	Number of indices to follow
i(1)	Index of Level-1 ancestor. Always = 1, for **OcaObject**.
i(2)	Index of Level-2 ancestor, if any.
...	
i(n-1)	Index of immediately prior ancestor (i.e. parent)
i(n)	Index of the class within the given parent.

Index values need not be consecutive, nor need the sets of them be compact.

Datatype of **n** is **OcaUint16.** Datatype of **i1 ... i(n)** is **OcaClassIDField.**

Format of **OcaClassIDField** is:

1 bit	**Proprietary flag**.	1 for proprietary classes, 0 for standard classes.
15 bits	**Index value**.	Must be unique within the given parent, at each level.

Considering the entire 16-bit field as an unsigned integer, the following value range rules apply:

0	**Reserved.**	Will never be used.
1 ... 4095	**Reserved.**	For managers and other objects with fixed, predefined object numbers.
4096 ... 32767	**Standard classes.**	For standard OCC worker and agent classes.
32768 ... 65279	**Proprietary classes.**	For proprietary classes in production devices.
65280 ... 65535	**Proprietary classes.**	For experimental proprietary classes. For testing only.

Proprietary values may be assigned at the discretion of the manufacturer. Standard values will be assigned by the OCA standards organization.

If, in a Class ID {n, i1, i2, i3, ...} a particular index i(k) is proprietary, then all indices to the right of it must be proprietary as well. This is equivalent to saying that a standard class may not inherit from a proprietary one.

Note that OCA does not prevent two manufacturers from using the same proprietary class index value for different objects. This is not expected to be a problem, since controllers of proprietary objects will normally know the manufacturers of the devices they control, and behave accordingly. That said, it is imaginable that a device will need to integrate two disjoint proprietary classes with the same class ID. In this case, one of the two class IDs will have to be changed. Since there is a relatively large address space for proprietary class IDs, this change should be straightforward.

3.1.2.2. Methods, Properties, and Events

Like classes in programming languages, OCA classes have elements which allow access to their data and operating states. In programming classes, these elements define storage references and procedure calls and callbacks. In OCA, however, class elements are used to define protocol exchanges between devices and controllers. Because OCA includes a number of different protocols (OCP.1, OCP.2, and so on), the detailed relationship between class elements and protocol elements varies. Nevertheless, the set of class elements is the same for every OCA protocol.

The elements of classes are follows:

- A class's control or monitoring data (i.e. the parameters that are visible on the control network) are represented by class Properties.
- The class provides appropriate Get() and Set() Methods that controllers may invoke to retrieve and change property values and class operating states.
- Classes also may define Events, which are callbacks that devices may initiate to inform controllers of specific occurrences. To receive events, controllers must first subscribe to them. See section Events and Subscriptions.

3.1.2.3. Method, Property, and Event IDs

In the class tree, every method, property, and event is not only assigned a name, but also locally unique ID of the form:

LLtNN

where

LL	is 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	is a type code: p for properties, m for methods, e for events.
NN	is a sequence number starting at 01 for each class..

for example

01p01	is the first property of a class defined at tree level 1. In this case, OcaRoot is the only class defined at level 1, so 01P01 is the first property of OcaRoot.
03m02	is the second method of a class defined at tree level 3. There are several classes defined at level 3 -- this ID would apply to the second method of any of these classes.

The idea behind these IDs is 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.

To see this approach in action, please see the example of class OcaGain in section 4.4.1.1.

3.1.2.4. Construction Parameters

Each class has an associated set of construction parameters, which are used when a new object is instantiated from the class. New object creation is allowed in a certain class of device known as a "fully configurable" device -- see section 4.1.

Construction parameters are discussed more fully in section 12.

3.1.2.5. The Power of Inheritance

As will be seen in sections below, the cumulative effect of inheritance in the class tree generally results in a very rich set of properties, methods, and events for ordinary objects. Although the definition of a class may often seem rather simple, that definition is in fact augmented by its entire ancestry.

3.2. Messages

Control and monitoring operations are implemented by messages passing between objects. The term “control message” (or just “message” where the context is clear) means an OCP protocol data unit (PDU) sent from one object in a particular device to another object, usually in different devices.

With rare exceptions, OCA messages are fully acknowledged, with informative status indications in the return messages. OCA does not use multicasting for messages, for reliability reasons.

OCA does define a "fast" message type that is used for noncritical traffic such as level meter updates. Fast messaging is not acknowledged and may be multicast, but cannot be used for regular commands. Fast messaging is part of the Subscription mechanism, described in section 5.

3.3. Note on Implementation

The fact the OCA protocols are described in the language of object-oriented design does not imply that implementations of those protocols must be programmed in an object-oriented style.

The implementable content of OCA protocol designs is a set of protocol data unit (PDU) formats and exchange rules. Exchange and processing of those PDUs may be programmed in any style, as long as, in the end, the correct data is exchanged.

Those working in larger environments will probably find it useful to construct programming class libraries that mirror OCA class trees, and use full object-oriented programming methods. Those working in smaller contexts may find older methods more appropriate.

Object-based or non-object-based implementation: the choice is up to the implementer.

4. Device Model

The device model is the control model for the device. In other words, it is the device as seen by OCA. The internal implementation structure of the device may or may not resemble its device model; the only requirement is that in OCA exchanges device must behave as if it had the structure given by its device model.

4.1. Device Configuration Types

The configuration of an OCA device may be fixed, pluggable, partially configurable, or fully configurable.

- Fixed means that the device has a permanently assigned object repertoire and signal flow topology, defined at time of manufacture.
- Pluggable means that the device's object repertoire and/or signal flow topology 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 means that the device's signal flow topology may be changed while it is online, via OCP commands.
- Fully configurable is a superset of partially configurable, with the addition that OCP commands may be used to create and delete objects inside the device.

Fixed and Pluggable devices are sometimes called static devices, because controllers can't change their configurations. Partially configurable and Fully configurable devices are sometimes called dynamic devices, because their configurations can be varied in operation.

Details of OCA configuration management are in sections 4.4.3.1 and 4.4.3.2.

Creation and deletion of objects is discussed further in section 12.

4.2. Object Addressing

Within an OCA device, each object (i.e. each instantiation of a specific class) is identified uniquely by an object number (ONo). Assignment of ONos is accomplished as follows:

- In fixed and partially configurable devices, ONos are assigned at time of manufacture.
- In pluggable devices, ONos are assigned at device setup time.
- In fully configurable devices, ONos are assigned at time of object creation. In such devices, ONo values will not normally be re-used when objects are deleted and recreated without an intervening device reset. If the ONo address space becomes exhausted, unused ONo values will be re-used, beginning with the oldest. Since ONo's are 32 bits long, re-use is unlikely.

In specific implementations, developers may find it useful to use specific object numbering schemes to optimize device performance and/or ease object management. This is fully permitted (or even encouraged) by OCA.

OCA makes only three rules about ONo values:

1. ONo values01 through 4096 are reserved by OCA for managers and possible other future objects that require predefined object numbers.
2. No two objects may have the same ONo.
3. No two ONos may refer to the same object.

ONos inhabit a flat address space of 2^{32} locations. Unlike some other media control protocols, ONos are not multi-field (i.j.k. ...) values based on some hierarchical decomposition of the device. This is a deliberate design choice intended to maximize protocol efficiency and to minimize device overheads in resolving object references, handling command responses, and managing errors. ONos are sent and received in every OCA command and response, so an efficient implementation is critical.

OCA does provide powerful methods (see Blocks, section 4.4.3) for hierarchical representation of device object structure. However, every object inside of these hierarchical structures, no matter how elaborate they may be, is still addressed by a simple 32-bit value which is global within the device.

On the user interface side, controllers may choose to allow users to reference devices by hierarchical names, e.g. "Channel(1).Left.EQ(1).Frequency". When this is done, it is up to the controller to map each name into the appropriate ONo and property.

4.3. Overview

Figure 4 shows the OCA device model. The boxed items are objects.

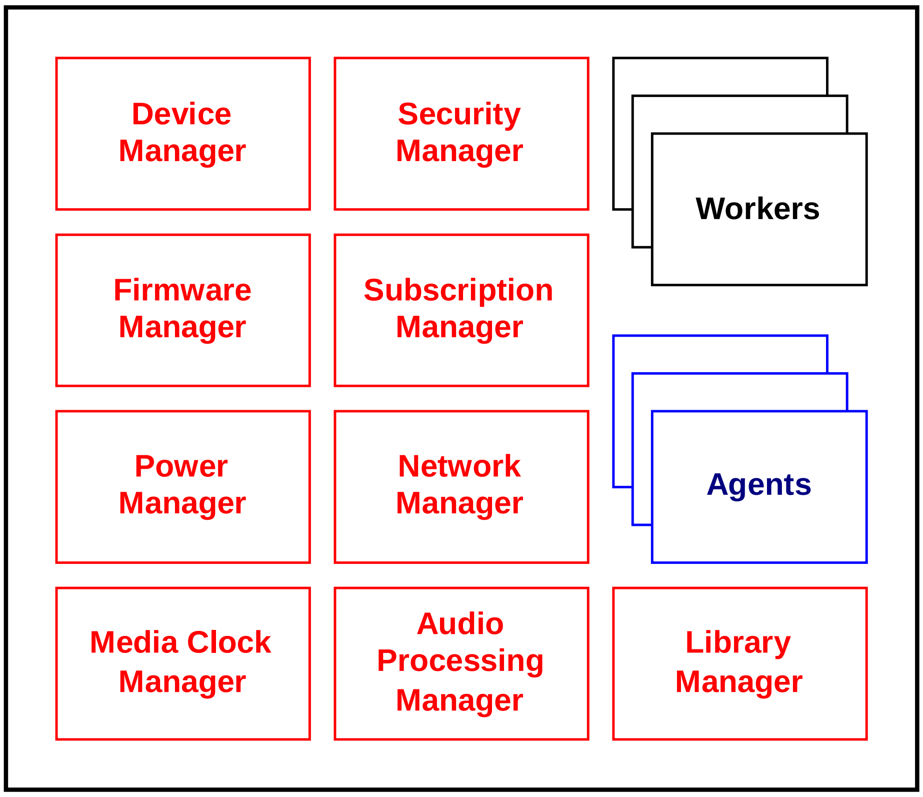


Figure 1. OCA Device Model

Elements of the device model are as follows:

- Managers

  Manager objects are control objects that affect or report the basic attributes and overall states of the device. There is only one instance1 of each manager class (i.e. one Device Manager, one Security Manager, etc.). Each manager has a standard, well-known object number defined in [OCC 1.1]. Some managers are mandatory, others are optional see section 4.6.

- Workers

  Worker objects are control interfaces to the device's application functions - audio mute switches, gain controls, equalizers, level sensors, overload sensors; video camera controls, signal properties, image processing parameters, et cetera.

  Workers are classified into Actuators, which control application functions (e.g., a switch); Sensors, which detect and report signal parameters and other values back to controllers

1 OCC refers to classes with only one instance as "singletons".

(e.g., a signal level sensor); Blocks and their associated Block Factories, which allow the grouping of related objects into collections; Matrices, which allow objects to be assembled into two-dimensional arrays; and Networks, which describe digital networks to which the device is attached.

- Agents

  Agent objects are intermediaries that affect control actions within the device. For example, there is an agent named Grouper that implements complex grouping of control parameters, in a manner that resembles VCA grouping in analogue systems.

  An agent does not itself represent a signal processing function, but instead changes a signal processing parameter in one or more associated workers.

Detailed definitions of all of the classes that define these objects are contained in the class structure document [OCC 1.1]. In this document, we summarize the available classes, giving examples and focusing on those with particular architectural significance.

4.4. Worker Classes

4.4.1. Actuators

Actuators control device signal processing and housekeeping functions. Details are in [OCC 1.1]. In any device, any actuator class may be instantiated as many times as required for control of application function.

[OCC 1.1] defines 36 Actuator classes. A few examples are:

• OcaGain	Controls a gain function.
• 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

4.4.1.1. Detailed Example: OcaGain

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, be they Workers, Managers, or agents.

- Lineage. OcaGain's class tree lineage is shown in Figure 2. Note that the topmost three classes - OcaRoot, OcaWorker, and OcaActuator -- are abstract classes that will never be instantiated per se. They are present in the class tree to express certain common characteristics of objects, workers, and actuators, respectively.

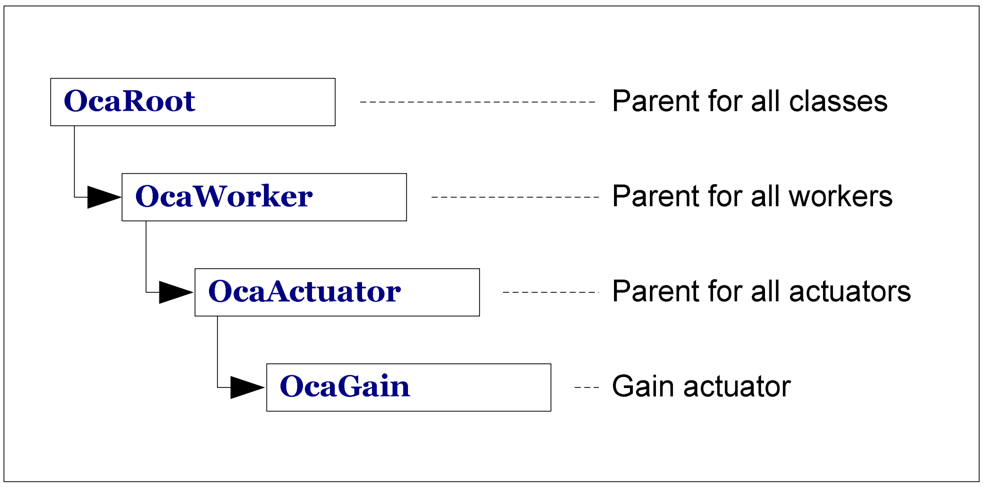


Figure 2. OcaGain Lineage

- 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 1. These properties are accessed via method calls, shown in Table 2. Events that may be raised by OcaGain are shown in Table 3.

  As the tables show, the repertoire of method calls is relatively large -- larger than simple devices may need. OCA allows for this by defining a "not implemented" status value which methods may return when they are not implemented in the device at hand.

Name	ID	Datatype	Ax	Description	Defined in
ClassID	04p01	Class ID	RO	Class ID of OcaGain class	OcaGain
Class Version	04p02	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, e.g. "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 structure s	DD	Collection of input and output ports that this object has	OcaWorker
Label	02p05	String	RW	Purpose of object in system, e.g. "Elvis Vocal Gain"	OcaWorker
Owner	02p06	ONo	RO	Object number of containing block	OcaWorker
Gain	04p03	Float	RW	The gain value in dB	OcaGain

Table 1. OcaGain Properties
("Ax" means "Access". "RO" : read-only; "RW": read/write, "DD"=Device dependent)

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 2. OcaGain Methods

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

Table 3. OcaGain Event

4.4.2. Sensors

Sensors detect the value of some parameter and transmit it back to controllers. Through the use of the OcaNumericObserver agent (see section 4.5.4), a sensor's reading may be transmitted conditionally (e.g. when it exceeds a defined threshold) or periodically.

[OCC 1.1] defines 20 Sensor classes. A few examples are:

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

4.4.3. Blocks

A Block is a special kind of worker that is able to contain other objects.

- It can contain workers.
- It can contain agents.
- It can contain other blocks.
- And it has a signal flow topology.

We refer to an object inside a block as a member of that block. We refer to the block which contains an object as the object's container.

Each block is described by a class (the block class). The base class for all block classes is named OcaBlock.

A block class does not represent specific audio processing in the way that other worker classes do. The block class simply represents a collection of workers, agents, and nested blocks, and the signal flows within that collection.

Blocks are central to a number of OCA mechanisms. It is recommended that the reader spend sufficient time on this section to feel comfortable with Block concepts and features.

4.4.3.1. Block Contents

Every OCA device has at minimum one block (the Root Block) that contains all of the device's workers, agents, and blocks. More advanced devices may contain nested blocks, to any level. Fully reconfigurable devices may allow the creation, modification, and deletion of blocks.

Every worker and every agent is a member of exactly one block. For simple devices, they will all belong to the Root Block.

Managers do not belong to blocks.

The object set of a Block is the set of workers and agents it contains. Thus, the object set of the Root Block contains all the workers and agents inside the entire device. Some of those workers and agents might be inside nested blocks.

The OCA representation of an object set is called an object list. OcaBlock offers methods which allow retrieval of the object list of the Block in question. These methods allow retrieval only of the directly contained objects, or full recursive retrieval of all directly contained objects plus all objects inside sub-blocks, sub-sub-blocks, et cetera.

For fully reconfigurable devices, OCA offers methods that allow controllers to modify a Block's object set, and to create and destroy whole blocks.

4.4.3.2. Signal Flow

4.4.3.2.1. Ports

For the purpose of defining signal flow, every worker has one or more signal ports (or just ports).

- A port is an input port if it represents a signal flow into the processing function the worker represents.
- A port is an output port if it represents a signal flow out of the processing function the worker represents.

4.4.3.2.2. Block Ports

Blocks themselves have signal ports, called block ports. Block ports are intermediate ports that exist to define connection points on the exterior surface of the block .

- A block input port is a connection point for signals entering the block. To objects inside the block, an input block port appears as an output (signal source) port.
- An block output port is a connection point for signals leaving the block. To objects inside the block, an output block port appears as in input (signal sink) port.

4.4.3.2.3. Signal Paths

A signal path is an internal connection between an output port and an input port in the same device.

- 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 disparate blocks.

A signal path's input and output ports will usually be on different objects, but OCA does not require this.

4.4.3.2.4. Signal Flow

The set of all signal paths with at least one end inside a Block is called the block's signal flow. The block's signal flow does not include signal paths that extend from block ports to other ports outside the block. Such paths belong to the containing block(s).

The OCA representation of a signal flow is called a signal flow list.

OcaBlock offers methods which allow retrieval of the signal flow list for the block in question. These methods allow either retrieval only of the directly contained signal paths, or full recursive retrieval of all directly contained signal paths and all signal paths inside sub-blocks, sub-sub-blocks, et cetera.

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

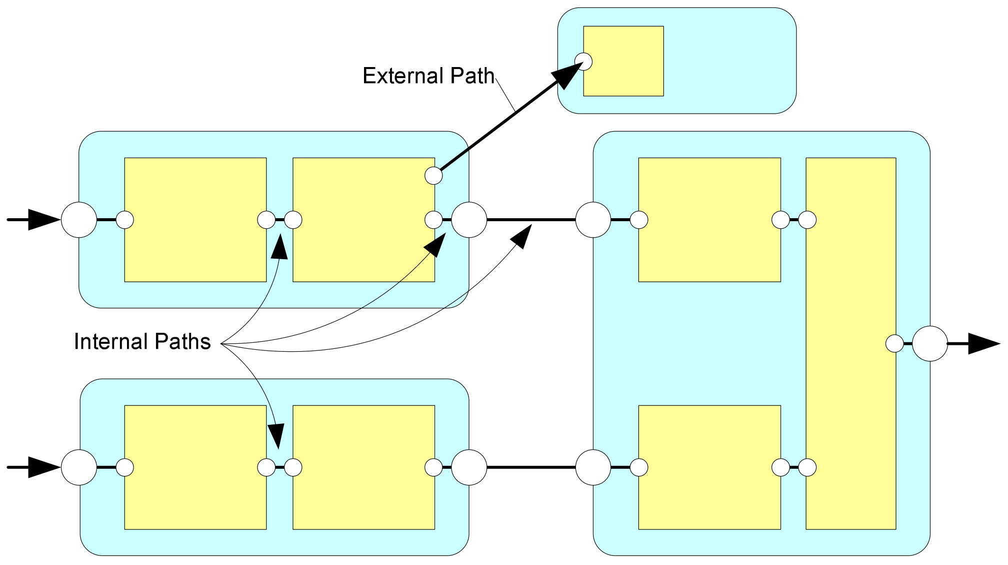


Figure 3. Signal Flow Diagram

Media transport connections between one device and another are not considered part of device signal flow. Inter-device media connections are described in section **Error! Reference source not found.**.

4.4.3.3. Abstract Nature of Blocks

It is important to remember that OCA 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. OCA makes no assumptions about how blocks are defined, nested, or interconnected. Blocks may be nested to any level.
- There is no hierarchical object addressing scheme based on block containment. Object numbers (see §4.2) are simple integers whose values are not semantically significant to OCA.
- For signal flow management, blocks offer a hierarchical interconnection scheme, but the use of it is optional. This is detailed in section 4.4.3.2.
- Blocks do not aggregate control functions -- i.e., they do not provide grouping or mastering. Such aggregation is provided by the OcaGrouper class. The OcaGrouper mechanism is independent of block boundaries, and fully supports multiple overlapping grouping functions -- see section 4.5.1.

4.4.3.4. Examples

Here are a few block examples. In the diagrams, circles represent ports. The larger circles represent block ports. Lines connecting circles are signal paths.

4.4.3.4.1. Simple Mic Channel

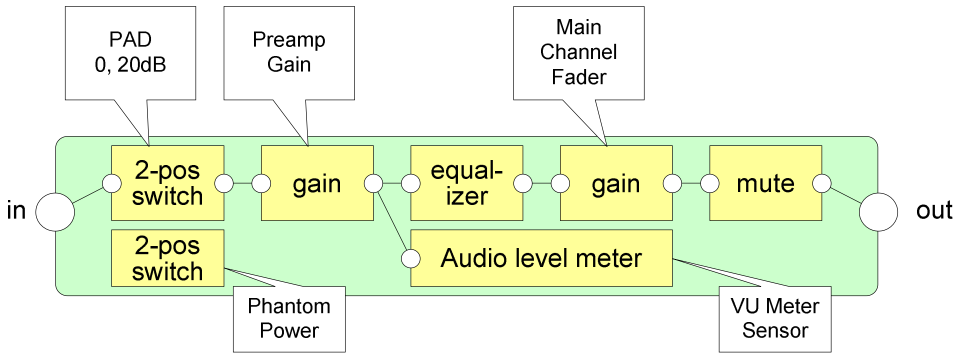


Figure 4. Simple mic channel

This example illustrates three OCA principles:

- Objects of the same class (e.g. two-position switch, gain) may be used for different purposes within the device.
- Not all objects (e.g. phantom power switch) need have signal connections.
- Sensors (e.g. audio level meter) do not have signal outputs.

To aid in managing objects, OCA supports the storage of two text strings inside each worker, as follows:

- The text property Role is a readonly string that gives the role of the worker in the device; e.g. "Preamp Gain".
- The text property Label is a read/write string that allows controllers to designate the function of the worker in the application context, e.g. "Elvis Vocal Gain".

4.4.3.4.2. Two-Channel Mic Mixer

This example illustrates the use of blocks in larger assemblies. The mic channel block of Figure 4 is replicated and combined with another gain control to make a simple mic mixer.

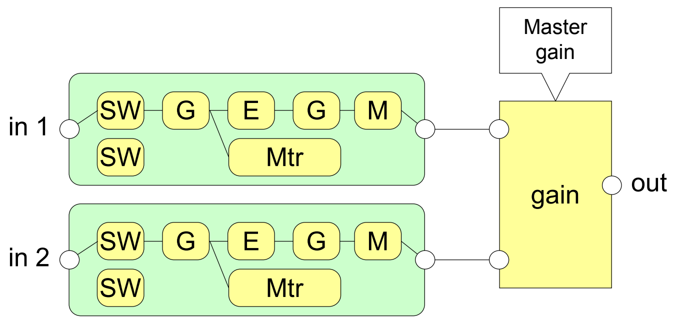


Figure 5. Two-channel Mic Mixer

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

Note that there is no explicit object for the mix bus and summing amplifier. The reason for this is another OCA principle:

- OCA only models the control and monitoring functions of a device, not its entire signal path. OCA seeks to offer a control and monitoring model only, not a complete device implementation model. If a function has no remote control features, OCA does not model it.

For this simple mixer, the mix bus / summing amplifier do not have any parameters that are remotely controllable. Therefore, they do not have a corresponding OCA object.

More sophisticated mixers might have control and/or monitoring functions associated with the summing subsystem. In that case, the OCA signal flow might include an explicit summing point.

4.4.3.4.3. Nested Blocks

We will now consider the equalizer object shown in the mic channel model in Figure 4. A basic mic channel equalizer might contain a high-pass filter section, followed by perhaps three parametric equalizer sections.

OCA defines a class that can be used to model a high-pass filter (class OcaFilterClassical), and another class that can be used to model a parametric equalizer section (class OcaFilterParametric). Thus, to model the mic channel equalizer, we will require one OcaFilterClassical instance, followed by three OcaFilterParametric instances.

We could just add these to the mic channel inline:

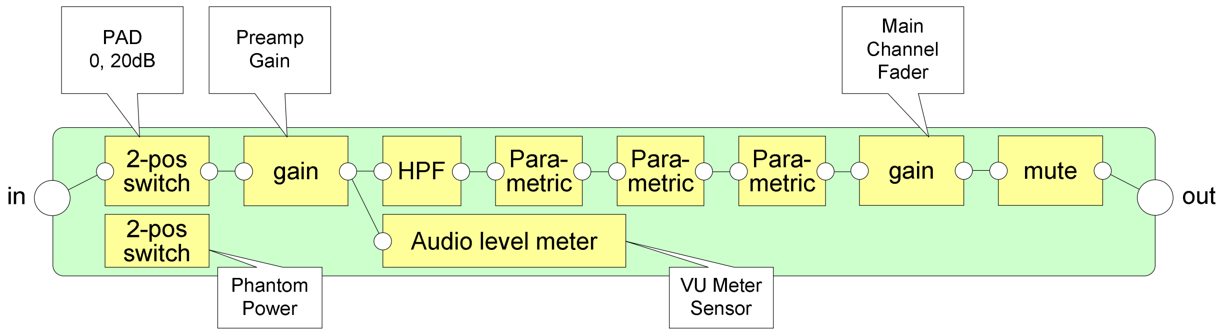


Figure 6. Mic channel with EQ sections inline

In this figure, the object labeled HPF is an instance of class OcaFilterClassical, and the objects labeled Parametric are instances of class OcaFilterParametric.

Alternatively, we could define a block named, say, MicEqualizer and nest it inside the mic channel block.

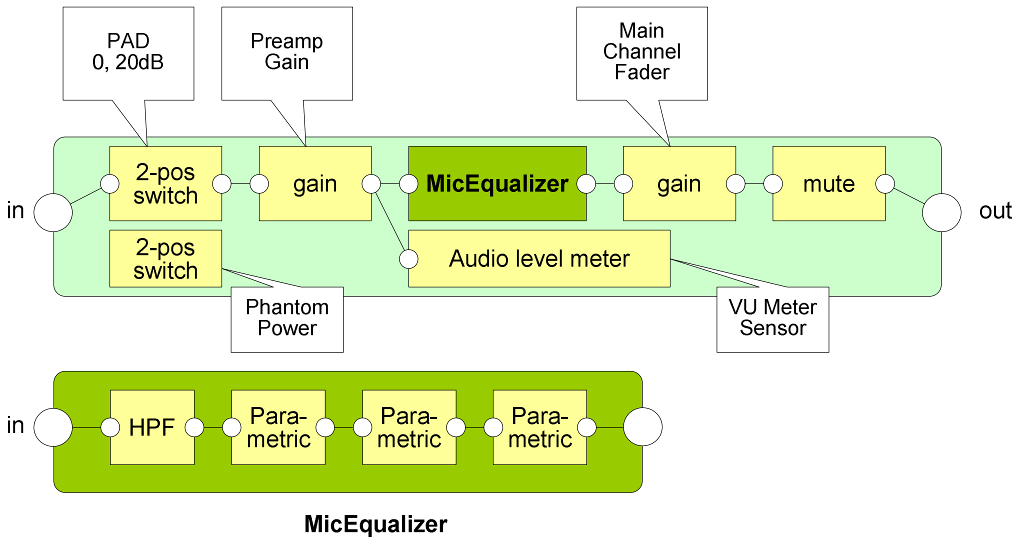


Figure 7. Mic channel with MicEqualizer block

This construct is tidier and will be easier to use in reconfigurable devices. Also, it might make controller implementation more straightforward, particularly if the same equalizer were used in various parts of the device, and/or in other products.

4.4.3.5. Block Factories

In fully reconfigurable devices, controllers must be able to build blocks, populate them with objects, and "wire" signal paths among those objects. OCA provides two ways of doing this.

1. The controller can instantiate an object of class OcaBlock, then send commands to the device which create and interconnect objects inside the new block. OR

2. The controller can use a block factory.

A block factory is a worker whose job is to construct fully populated and "wired" blocks. Each block factory is 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. Block factories are useful when controllers must create multiple blocks, all of the same kind.

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

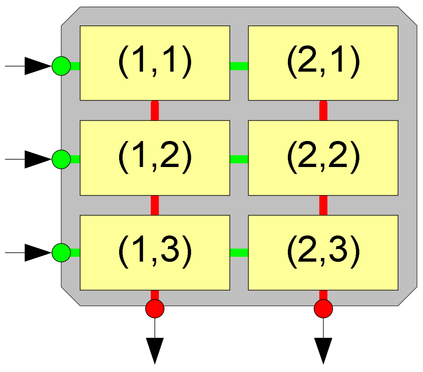
There are two ways of creating block factories. Either or both of these methods may be implemented, depending on device type.

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

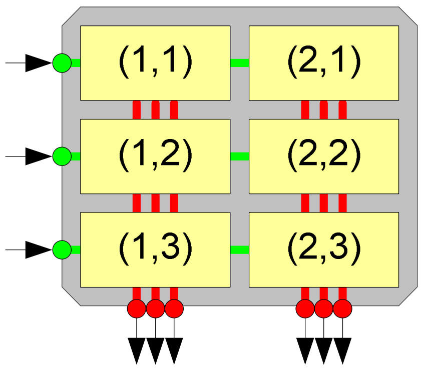
2. The controller may create block factories, using specific OCA commands to define the configurations of blocks they will create. In this case, controllers are free to define new block types which may be subsequently instantiated.

4.4.4. Matrices

The OCA matrix is a generalization of the familiar audio matrix concept. OCA matrices are rectangular arrays of identical worker objects that share one or more common input and output busses for the rows and columns, respectively.



1-in, 1-out



1-in, 3-out

Figure 8. OCA Matrices

If the workers are switches, the matrix might represent a conventional switching matrix; if the workers are gain controls, the matrix might represent a conventional mixing matrix. However, in OCA the workers can be of any class, even blocks. For example, we could construct a matrix of blocks such as shown in Figure 9 and Figure 10:

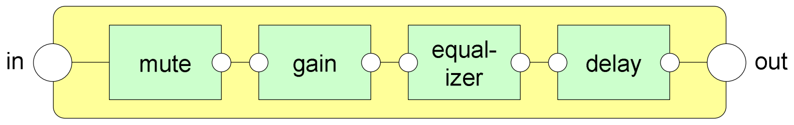


Figure 9. Example of complex matrix element

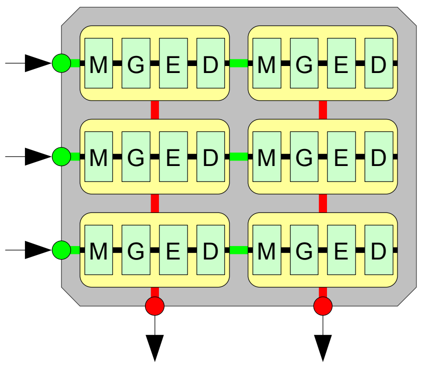


Figure 10. Matrix of complex elements

4.4.4.1. Matrix Addressing

As suggested by Figure 8, OCA matrix elements are addressed by coordinates. The OCA convention is to use (x, y) coordinates, where x is the horizontal (column) coordinate, and y is the vertical (row) coordinate.

Normal coordinate values range from 1 to the number of rows or columns.

The special value zero is used to denote the entire row or column. So, for example, (0,2) means the entire second row. The use of this feature will be shown in §4.4.4.3.

4.4.4.2. Matrix Structure

An OCA matrix is based around an object which is an instance of the OcaMatrix class, but that instance does not include the objects which constitute the matrix elements. Those are instantiated on their own, and are called the members of the matrix. Their class is called the matrix's member class. All of a matrix's members must be of the same class.

The members of an OCA matrix must reside in the same device as the matrix instance.

In addition to the matrix object and the members, the OCA matrix includes one additional instance of the member class, called the matrix proxy. The matrix proxy is used by controllers to access the settings of matrix members via coordinates.

A complete matrix is shown in Figure 11.

4.4.4.3. Accessing Matrix Elements

Accessing a property of a matrix element is done in two steps:

1. The controller calls the OcaMatrix method SetCurrentXY(x, y), specifying a target coordinate set.

2. The controller calls the matrix proxy's Get or Set method for the desired property. If Set is called, the new value of the property is specified.

   The special values x=0 and y=0 may be used in conjunction with the Set methods to perform aggregate set operations as follows:

SetCurrentXY() Call	Set<property>() Action
SetCurrentXY(0, y)	New value is set in all objects of row y
SetCurrentXY(x, 0)	New value is set in all objects of column x
SetCurrentXY(0, 0)	New value is set in all objects of the whole matrix

   Using x=0 and y=0 with matrix Get methods is not supported.

OCA recognizes 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 the legal range, it is rejected with an appropriate error indication.

The SetCurrentXY(x,y) method is used when controllers wish to access matrix members using (x, y) coordinates. Matrix members may also be accessed directly, via their object numbers, should this be required. When this occurs, the matrix mechanism is bypassed but not damaged.

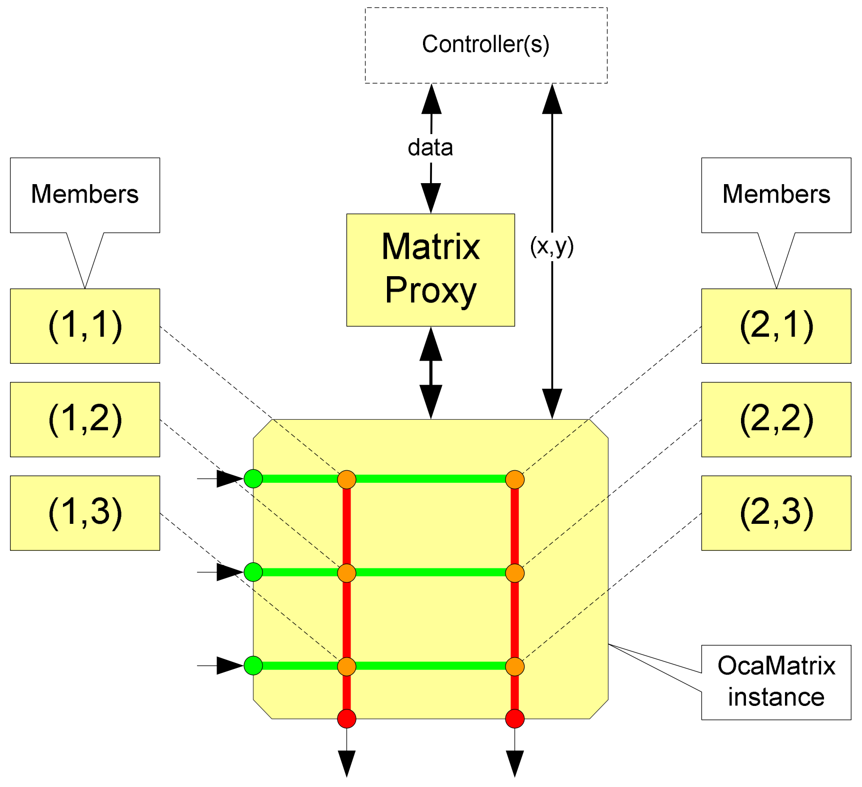


Figure 11. OCA Matrix Structure

4.4.4.4. Matrix Signal Flow

A matrix has input and output signal ports (matrix ports) that are separate from the input and output signal ports of its members.

- Matrix rows correspond to matrix input ports; matrix columns correspond to matrix output ports.
- Matrices may have one or more ports per row or column.
    - A matrix of (N) rows has (N * PPR) row ports, where PPR is the matrix's number of ports per row. Row ports are matrix inputs.
    - A matrix of (M) columns has (M * PPC) column ports, where PPC is the matrix's number of ports per column. Column ports are matrix outputs.

When a worker becomes a member of a matrix, its input and output ports are connected to matrix row and column ports in corresponding order. Specifically:

- Member input port 1 is connected to row port y, where y is the matrix row to which the worker belongs.

- Member output port 1 is connected to column port x, where x is the matrix column to which the worker belongs.
- If PPR is greater than 1, member input ports 2 through PPR are connected to row ports y+1 through y+PPR-1, respectively.
- If PPC is greater than 1, member output ports 2 through PPC are connected to column ports x+1 through x+PPC-1, respectively.

Members may define additional ports not used by the matrix; these may be connected directly to other ports in the device as needed, bypassing the matrix ports.

4.4.4.5. Summing

Looking at Figure 12, it is evident that multiple member output ports are connected to each matrix output port. The implication is that these outputs are summed to create the final matrix output signal. This implied summation is defined for matrices only. In all other contexts, signals flows which connect multiple signal sources to a single sink are not permitted.

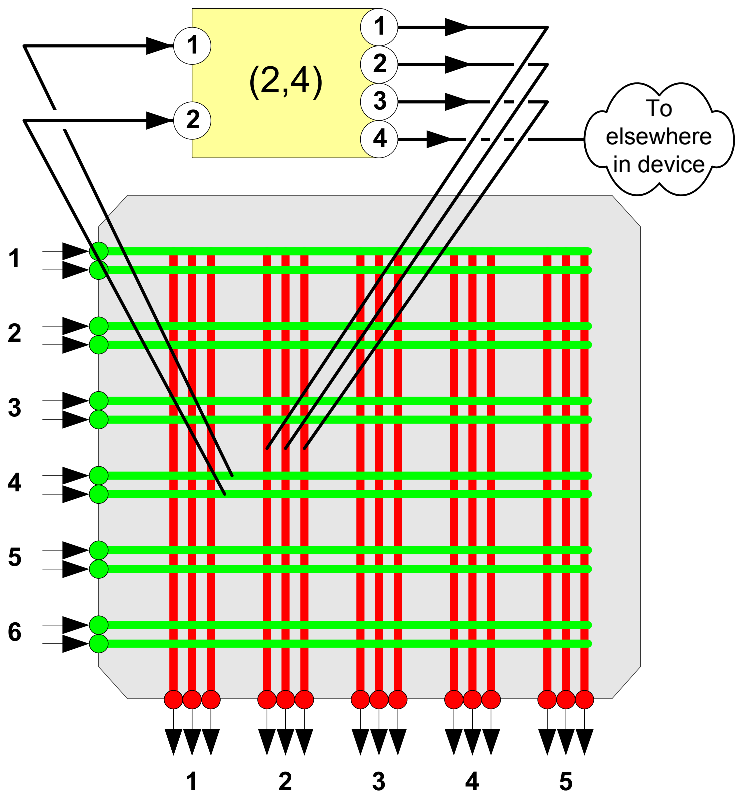


Figure 12. Matrix Port Relationships.
Ports per row = 2, ports per column = 3.

4.4.4.6. Applications of OCA Matrices

Because they are defined rather generally, OCA matrices may find use not only for the representation of traditional audio matrixing facilities, 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 multichannel 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 bus which could be represented by a matrix row port. However, they would have separate output ports, so no matrix output ports would be defined.

4.4.4.7. Matrix Deployment

OCA matrices and their proxies must be instantiated in the same device as their members. For an aggregation function capable of spanning devices, please see OcaGrouper (§4.5.1).

4.4.5. Network Worker

OCA 1.1 defines one network Worker, named OcaNetworkMediaPort. It is part of OCA's general networking mechanism, and will be described in §4.5.1.3, below.

4.5. Agent Classes

OCA 1.1 defines seven agent classes. Each of these implements a notable OCA mechanism, as described in the following sections.

4.5.1. OcaNetwork

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

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

4.5.1.1. Network Classes and Objects

OCA network support is anchored in the Network Manager (class OcaNetworkManager), a singleton object that is present in all OCA devices.

The Network Manager links to one or more Network agents. Each Network agent represents one network to which the device is connected. The network may be a control and monitoring network, a media transport network, or a combination.

The root class for Network agents is OcaNetwork. The specific implementation of this class will depend on what kind of network is to be represented. The interface of OcaNetwork does not depend on the type of network being represented, but some implementations may need to create more specific subclasses of OcaNetwork in order to represent their network types adequately.

A configuration of OCA for a particular network type is called an OCA *Adaptation*.

Two Network agents can refer to the same physical network. For example, an OCA Adaptation for a network that provided TCP/IP for control and monitoring, but used the IEEE 1722/1722.1 protocol suite for media transport, would entail two Network agents, one for IP control traffic, and another for 1722 media stream(s).

4.5.1.2. General Features

Each OcaNetwork instance provides the following features for the network it represents:

- Functions to start up, pause, and shut down the interface for the network.
- Standardized network status indication.
- Ability to get and set the host name or ID by which the device advertises itself on the network.
- Identification of the hardware link type (TCP/IP Ethernet, USB, etc.) of the network.
- Identification of the system interface the network uses for input and output.
- Identification of the control (if any) and media transport (if any) protocols the network uses.
- Transmission performance and error statistics.

4.5.1.3. Media Transport Management

OCA defines an interface for the management of media transport connections. Through this interface, OCA controllers can access media network connection management functions to set up, monitor, and tear down media transport connections between devices.

An OcaNetwork agent for a media transport network includes two collections, as follows:

- *Media Ports*. A media port is a Worker object that is capable of being bound to an external media stream arriving from or departing into the media transport network. Through media ports, the objects inside the device access the signals in media streams.. Media ports are instances of the class OcaNetworkMediaPort.
- *Media Transport Connections*. A media connection is the binding of a media port to an external media transport stream for the purpose of sending or receiving media stream data. OCA supports both single-channel and multichannel media transport connections.

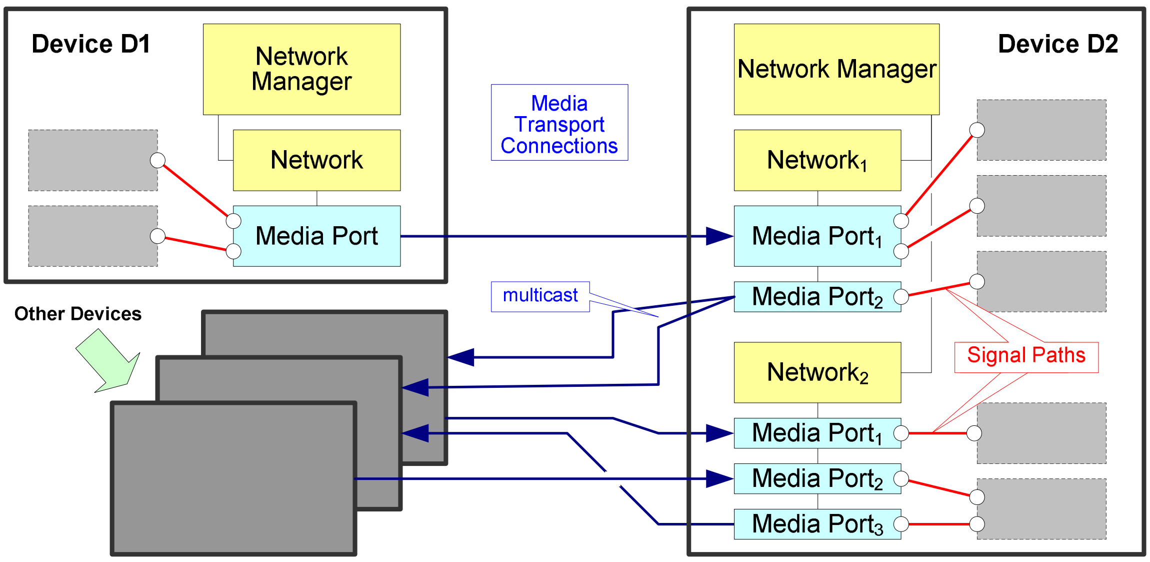


Figure 13. Media transport connections

The Media Port object includes methods which allow controllers to create and delete media transport connections. As well, the Media Port object informs subscribing controllers of media transport connections that have been made pursuant to requests from external devices.

The Media Port object includes a feature for identifying a media transport connection as a secure one. However, the actual implementation of media transport security is left up to the specific network implementation.

OCA places no restrictions on media stream formats, sampling rates, encodings, et cetera.

4.5.2. OcaGrouper

A grouper is an object responsible for aggregating property values. By "aggregating", we mean associating property values in such a way as to make them controllable as a single value. Aggregation supports audio control functions variously known as ganging, linking, mastering, submastering, and VCA mastering.

No signals pass through groupers - they affect control flow only. In this respect, they resemble traditional VCA masters more than traditional mix groups and subgroups.

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

An actuator grouper allows control of many actuator objects from a single input value; a sensor grouper allows observing many sensor objects via a single output value. Actuator groupers are described below; sensor groupers are TBD.

OCA groups aggregate whole objects, not individual properties, but still maintain the distinction between different property types. For example, the OcaFilterParametric class represents a parametric equalizer with several properties - frequency, Q, and passband gain. When a grouper groups a set of OcaFilterParametric objects, all of these properties are grouped, but the group parameters are kept separate from each other.. Thus, the grouper would provide separate group parameters for each of frequency, Q, and passband gain.

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

4.5.2.1. Multiple Group Membership

In a working media system, many objects will be members of multiple groups. For example, in a multiway stereo sound reinforcement system, the left channel woofer power amplifier's gain control object might be controlled by a master gain group, a left-side gain group, and a woofer gain group.

For a property that belongs to an object which is a member of two or more groups, its setpoint will depend on the cumulative effect of the corresponding group setpoints of all the groups of which it is a member. Because all properties have range limits, it is possible that the cumulative effect of all applicable groups might drive a property out of range. This effect must be prevented or at least managed. Therefore, the grouper must know about all the groups and which objects belong to which, and must have algorithms which use this knowledge to forestall range limit issues.

4.5.2.2. Group Structure

As noted above, the grouper must know about all the groups and all of the objects which belong to them. We use the following terminology:

• Citizen	An object of which a grouper is aware.
• Group	A group of which a grouper is aware.
• Enrollment	The binding of a citizen to a group.
• Member	A citizen that is enrolled in a group.

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

A grouper holds references to its citizens, but does 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.

For each grouper, all the citizens must be instances of the same class.

Each group is accessed via a proxy object, called the group proxy. To change a group setpoint, a controller changes the corresponding property of the group proxy. Group proxies are automatically instantiated and deleted by the grouper, and always reside in the same device as the grouper. A grouper that manages (N) groups will have (N) group proxies. The class of a group proxy is identical to the class of the group's members.

Figure 14 shows a typical grouper configuration for controlling a stereophonic set of three-way loudspeaker systems.

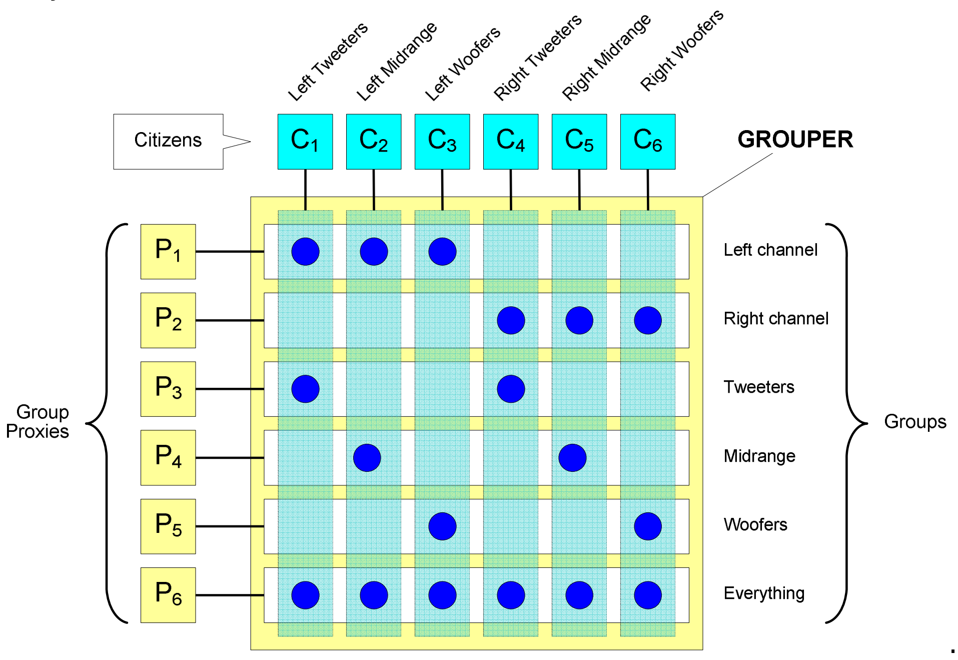


Figure 14. Typical Grouper.
Squares are objects, circles are enrollments.

A grouper does the following:

- Creates / deletes all the groups and group proxies.
- Registers / de-registers all the citizens.
- Enrolls / de-enrolls citizens in / from groups.
- Computes and sets new property values when group setpoint values change.
- Manages overrange and underrange conditions proactively, so that citizens are not asked to assign out of range values to their properties.
- Handles error conditions that arise if / when connections between grouper and citizen are lost.

4.5.2.3. Aggregation and Saturation Rules

Citizens have aggregation rules, which determine the algorithms by which their setpoint values are computed. Groups have saturation rules, which control handling of overrange conditions. These rules will vary from citizen class to citizen class, property type to property type, and product to product.

4.5.2.3.1. Basic Aggregation Rules

In theory, groupers can be implemented with a wide range of algorithms for calculating individual citizens' setpoints. Appropriate kinds of aggregation rules depend on the datatypes of properties being grouped.

For continuous-value parameters such as gain and delay, the SUM rule is frequently used:

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

where an "applicable" group is one to which the citizen belongs, and offset is a value private to each citizen. Offset records any citizen-specific settings that make its setpoint different from the group's setpoint. It may be thought of as a citizen-specific "trim" setting.

For discrete-value parameters such as selector switch settings, the LINK rule is frequently used:

citizen setpoint = most recently changed applicable group setpoint

For boolean parameters such as mutes, boolean rules such as AND, OR, and XOR may be useful besides the default LINK rule, e.g.

citizen setpoint = XOR(setpoints of applicable groups)

For groupers, the definition problem is complex, because objects with multiple properties may be grouped, and the aggregation rules will usually differ among the properties.

The basic OcaGrouper class defines a default set of aggregation rules (see §4.5.2.3.4), but it is expected that applications will need to define children of OcaGrouper to fulfill their own specific aggregation needs.

4.5.2.3.2. Basic Saturation Rules

There are two basic saturation rules:

- Saturating. The grouper allows all setpoint changes, but truncates values sent to individual citizens so that out-of-range conditions are narrowly avoided. When group setpoints return to more midrange values, normal operation of the aggregation rule resumes.

- Nonsaturating. The grouper refuses to make any setpoint changes that would force any citizen's property out of range.

4.5.2.3.3. Connection Loss Handling

When groupers lose contact with their 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 in which group setpoint changes have been made.

For situations such as these, application-specific processing may be required in order to provide the required behavior. OcaGrouper does not attempt to define all possible connection loss handling semantics. It is expected that devices will define subclasses of OcaGrouper which implement the specific behavior required.

4.5.2.3.4. OcaGrouper Defaults

OcaGrouper is a base class that defines common concepts and terms for groupers, a canonical control interface, default saturation and aggregation rules, and default connection loss handling. However, these features will vary from case to case. It is expected that specialized grouper subclasses will be common in practice.

The default rules are as follows:

- Aggregation: Depends on the base datatype of the grouped property. The "base datatype" of a property is its underlying machine datatype. Defaults are as follows:

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

- Saturation: Nonsaturating.
- Connection Loss Handling:
  1. OcaGrouper defines an event named StatusChange to which controllers may subscribe in order to be informed of citizen connection losses.
  2. After a connection is lost, the grouper will continue to control the citizens that it can reach. If the lost citizen becomes available again, the grouper will update its

property values appropriately. This will restore consistency, as long as the lost citizen's setpoint values did not change during the time when it was unavailable.

Note: this default represents a minimum safe behavior that may not suit all applications. Devices requiring a different behavior may create subclasses of OcaGrouper with the required behavior. It is expected that such subclassing will be relatively common in practice.

4.5.2.4. Grouper Deployment

Groupers may be instantiated in ordinary devices, or in dedicated OCA control servers, or in system controllers. The choice among these is a system design decision. OCA will support a flexible range of deployment scenarios.

4.5.3. OcaRamper

OcaRamper defines a mechanism (ramper) by which controllers can cause devices to execute incremental and/or prescheduled parameter changes. The rationale for rampers is:

- It is usually impractical to implement incremental changes (fade-ins, fade-outs, and crossfades) by sending sequences of OCA commands over the network, because the amount of control traffic required would be excessive.

- It is often not possible to implement prescheduled parameter changes in controllers, because application systems may need to execute such changes when controllers are not running.

- In general, timing accuracy of ramping actions will be more precise if not subject to network delays.

Each ramper is bound to a particular worker property. Ramper principal parameters include target property value, ramp duration, ramp start time, and ramping interpolation law.

Each ramper maintains a state property whose value can be monitored by controllers to determine ramping status - waiting to start, in process, complete, aborted, et cetera.

4.5.4. OcaNumericObserver

OcaNumericObserver is a property value "watcher" object that monitors the value of a specified numeric property in another object and, under certain conditions, notifies controllers of the value. It is part of the OCA Event and Subscription mechanism, and is described in section 5.2.

4.5.5. OcaLibrary

OcaLibrary defines mechanisms for the handling of parameter value sets that are pre-stored in devices. Such parameter sets have historically been called "presets", "patches", "memories", and other things.

OCA's features in this area are organized around the Block mechanism (§4.4.3). OCA defines two kinds of prestored parameter sets:

1. A ParamSet is a prestored set of property values for one particular class of Block.

- The act of installing these values into an instance of the Block is called applying the ParamSet to the block.
- The application of a specific ParamSet to a specific Block instance is called a ParamSet Assignment, or just Assignment when the context is clear.
- A ParamSet need not contain values for all of the properties in its target Block. Indeed, it might contain as little as one value.
- Applying one ParamSet to a Block does not necessarily erase the effect of previous ParamSets. If one ParamSet affects different properties from another one, then both will continue in effect.
- The block class for which a ParamSet is defined is called the ParamSet's target block class.
- A single ParamSet may be applied to any number of blocks, as long as all of the blocks are instances of the ParamSet's target block class.
- Read-only properties (e.g. ClassID) will not be saved or restored by this mechanism.

For example:

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

2. A Patch is a collection of Assignments.

   - The act of executing all the Assignments in a Patch is called applying the Patch to the device.

The entire library feature is optional. OCA devices need not implement any library-related objects or functions.

4.5.5.1. OcaLibrary Structure

A collection of ParamSets is called a ParamSet Library. A collection of Patches is called a Patch Library. OCA allows a device to have any number of ParamSet Libraries and any number of Patch Libraries.

Each library in a device is represented by an OcaLibrary instance. All of these instances, both for ParamSet and Patch Libraries, are anchored in the OcaLibraryManager object.

The binary format of a ParamSet is device-dependent and is not defined by OCA. Within the OCA protocol, a ParamSet is treated as an undifferentiated binary large object (BLOB). OCA includes primitives for upload, download, creation, and management of ParamSets, but these primitives do not include functions that "look inside" those ParamSets.

4.5.5.2. Creating ParamSets

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

1. A controller may download a ParamSet to a designated library in the device. . OR

2. The controller may request a block to save all of its objects' property values as a ParamSet in a particular library in the device. This "snapshot" action captures the values of all the properties of all the objects inside the block and saves them as a ParamSet in the designated library.

4.5.6. OcaMediaClock

OcaMediaClock describes a particular internal or external media clock which the device uses. It includes features for specifying clock source, rates, lock states, and so on.

OCA allows a device to define any number of media clocks, each one represented by its own instance of OcaMediaClock. All of these instances are anchored in the Media Clock Manager (class OcaMediaClockManager).

The media clock feature is optional. If an OCA device has no network-controllable clocking features, it need not instantiate OcaMediaClockManager or OcaMediaClock.

4.5.7. OcaEventHandler

OcaEventHandler is a special class used in the processing of OCA events. It is described in section 5.

4.6. Manager Classes

Table 4 shows all of the OCA manager classes. The Rqd or Opt column shows which managers are required for all OCA-compliant devices, and which are optional. Note that for required managers, not all of the parameters need be implemented for OCA compliance. Details are on [OCC-MIN 1.1].

As noted above, each manager class is instantiated at most once per device, and with a well-known object number.

Obj No.	Class Name	Class ID	Rqd or Opt	Function(s)
1	OcaDevice Manager	1.3.1	Rqd	Manages information relevant to the whole device - model and serial number, device name and role, overall operating state, device update lock, and other items.
2	OcaSecurity Manager	1.3.2	Rqd	Manages security keys.
3	OcaFirmware Manager	1.3.3	Rqd	Performs firmware updating.
4	Oca Subscription Manager	1.3.4	Rqd	Manages subscriptions, the constructs by which devices inform controllers of significant events. See section 5.
5	OcaPower Manager	1.3.5	Rqd	Manages device power state, including multiple power supplies, battery supplies, and so on.
6	OcaNetwork Manager	1.3.6	Rqd	Manages the control and media transport network(s) to which the device is connected.
7	OcaMedia ClockManager	1.3.7	Opt	Manages media clock selection and control.
8	OcaLibrary Manager	1.3.8	Opt	Manages device patches and presets, the OCA elements that handle prestored device configurations and parameter settings. See section 4.5.5.
9	OcaAudio Processing Manager	1.3.9	Opt	Manages global audio-related parameters such as headroom and sampling rate.

Table 4. OCA Manager Classes

5. Events and Subscriptions

In the control and monitoring of signal processing and routing, it is often necessary for devices to automatically report the values of varying parameters on a continuous basis. OCA provides for this function via the subscription mechanism, in which certain objects can be made to transmit property value update messages to other objects automatically and repetitively.

Objects that support subscriptions are called emitters, and the messages they emit to announce events are called notifications.

OCA includes specific features to address three particular issues in the processing of subscriptions:

a. Efficiency. OCA defines two types of subscriptions: reliable, in which notifications are transmitted via the normal OCP command communication mechanism; and fast, in which notifications are transmitted via less reliable but more efficient means. For example, in OCP.1 reliable notifications are sent via TCP, while fast notifications are sent via UDP unicast or multicast.

b. Multiple subscriptions per emitter. Emitters can support multiple simultaneous subscriptions. In theory, there is no limit to the number of such subscriptions. In practice, there will be implementation-defined limits that will vary from device to device.

c. Abandoned subscriptions. An abandoned subscription is a subscription whose subscriber has crashed or disappeared. When an OCA device detects a subscriber failure, it will delete all subscriptions which were made by the failed subscriber. In most OCA protocols, such detection will be done by sensing of keepalive messages.

Subscriptions are created by the Subscription Manager in response to controller requests. Subscriptions persist until the controller deletes them or until they are abandoned.

A subscription is a binding between two elements, an event and an event handler.

- An event is a specific event type of a specific object.

  For example, if object number 227 defines event types Start and Stop, then {227,Start } is an event, and {227,Stop} is another event.

- An event handler is an OnEvent method of an instance of the class OcaEventHandler.

  The object that owns the event handler is called the subscriber.

- The relationship between events and event handlers is many-to-many: an event may have any number of subscribers, and an event handler may participate in any number of subscriptions.

When a subscribed event occurs, the subscription mechanism transmits a notification to the event handler. This notification is in effect a call to the event handling method -- i.e., to the OnEvent method of the subscriber.

The notification message contains information which allows the event handler to understand the nature and context of the event, and to perform appropriate processing.

5.1. The PropertyChanged Event

OCA does not have a large repertoire of different event types. The small repertoire is sufficient mainly because the root class OcaRoot defines an event named PropertyChanged which, when used, fires whenever any of its object's property values changes.

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

PropertyChanged subscriptions can be used for such purposes as:

- Monitoring signal-induced object state changes;
- Monitoring overall device state;
- For devices with front panels, synchronizing controllers with user adjustments to front-panel controls;
- In multicontroller systems, synchronizing parameter changes among the various controllers.

5.2. Use of Numeric Observers

Sometimes it is efficient for devices to provide a tiny intelligence for monitoring parameter values. There are three common cases for this:

. There are three common cases for this:

- When monitoring a numeric or logical parameter, the controller wants notification only whenever the property value meets a certain simple numeric test -- for example, when it exceeds a defined threshold.; or
- A controller desires a regular periodic readout of a numeric parameter, regardless of its value; or
- The combination of the above: a controller desires regular periodic readouts, but only when the parameter value meets a certain value criterion.

OcaNumericObserver objects ("Numeric Observers") provide these capabilities. Each OcaNumericObserver object watches a specified parameter value, and emits an event named Observation whenever its criteria are met.

In application:

1. A numeric observer is created, given its criteria -- numeric test and/or periodic repetition rate -- and bound to the property of interest (the "observed property").

2. The controller subscribes to the numeric observer's Observation event.

Subsequently, whenever conditions meet the numeric observer's criteria, an Observation event is transmitted to the controller's event handler.

Numeric observers will most commonly be used in conjunction with sensor objects (e.g. audio level sensors), but can used equally with any property of any object. For example, it would be possible to define a numeric observer that emitted a notification whenever someone raised the gain of a power amplifier above some threshold level.

For example, Figure 15 shows use of a numeric observer to implement a signal-present light in a controller.

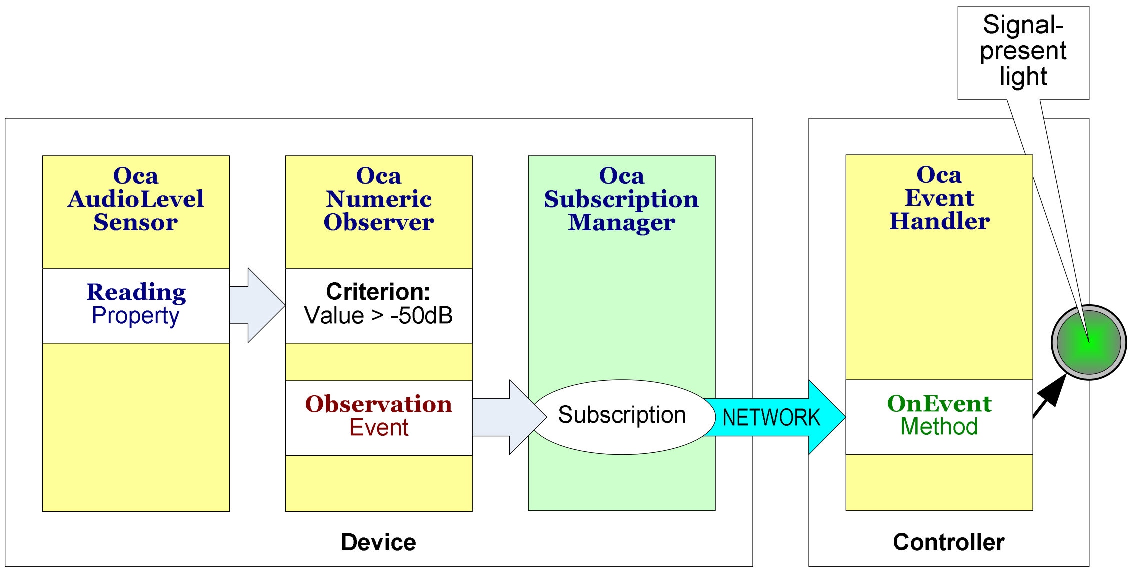


Figure 15. Using a numeric observer to implement a signal-present light

6. Sessions

OCA protocols are session-oriented, for at least the following purposes:

a. OCA requires request/response protocols - requests and responses have to be managed as connected pairs.

b. OCA objects need to have permanent application relationships with other OCA objects.

c. Discovery processes require enumeration sequences which extend over multiple PDU exchanges.

d. Prompt discovery of device failure is required.

e. Devices must be able to report changing parameters (e.g. signal level) to subscribing controllers on a regular basis.

f. These connection-like relationships must gracefully survive or gracefully vanish when transport interruptions occur.

OCA is required to support networks containing thousands of OCA devices. A central controller that has domain over hundreds or thousands of devices may not have the capacity to maintain a separate transport connection with each one. In such cases, controller hierarchies must be implemented, with successive levels aggregating control functionality in ways appropriate to the applications. OCA contains aggregation features -- notably the OcaGrouper class (§4.5.1) -- that will aid such implementations.

7. Security

OCA control security is a protocol-dependent element. In OCP.1 networks, security of the control data is handled by using the TCP/IP family's Transport Layer Security (TLS) protocol. Details are in [OCP.1 1.1].

Fully implemented OCA networks are capable of operating at levels of security sufficient to satisfy:

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

The OCA security scheme includes authentication and encryption, but does not include 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, it is up to the application to implement it. OCA security allows implementation of a secure access control mechanism and assures that, once access rights are established, the control and monitoring actions that follow will not be compromised.

Security data can be turned on or off globally. Networks with a mix of secure and insecure devices are not supported.

8. Concurrency Control

OCA supports the use of multiple simultaneous controllers with overlapping control domains, -- i.e., applications in which a particular object may receive directives from more than one controller. Because certain application control events require the exchange of more than one control message, the potential for a race condition exists.

To prevent race conditions, OCA devices support single-threading via a simple object locking mechanism with the following properties:

a. An optional locking mechanism is defined for all OCA objects.
b. An object may be implemented as lockable or non-lockable.
c. A lockable object may be locked by one remote object (the “lockholder”) at a time. A non-lockable object returns failure status for all lock requests.
d. The effect of the lock is to prevent all access to the object by anyone other than the lockholder.
e. Locks do not survive device resets.
f. When a controller's communication with a device fails for any reason, the device automatically removes all locks set by that controller. Controller to device communication is continuously monitored -- see section 9.1

OCA does not include a full lock management function. There is no deadlock detection, there are no non-exclusive locks, there are no multi-level locking schemes, et cetera. These functions, if needed, are left up to the controller application.

The lock function is implemented in each OCA class, as a set of methods and properties inherited from the base class.

It is possible to lock an entire device by locking its Device Manager object. This is only allowed when none of the device's other objects are locked.

9. Reliability

"Reliability" means the cumulative effect of Availability and Robustness. Fully implemented OCA networks are capable of operating at levels of reliability sufficient to satisfy:

a. international regulations governing emergency evacuation systems.

b. commercial and government uptime and robustness requirements.

c. public media and live performance uptime and robustness requirements.

9.1. Availability

High availability of OCA networks is supported by:

a. Continuous device supervision, via periodic "keep-alive" messages defined as part of the OCP protocol family.. Devices can also monitor themselves by means of a local OCP message that goes through the complete path from application to network and back.

b. OCP features for efficient network reinitialization following errors and configuration changes.

9.2. Robustness

To support robust implementations, OCA includes mechanisms for operation confirmation, and has fault-tolerant mechanisms that are resistant to PDU losses and device failures.

OCP protocols may use network type specific robustness mechanisms. For example, OCP.1 can take advantage of spanning-tree protocols to increase resistance to network link failures.

10. Device Reset

For recovery from terminal errors, OCA supports the device reset operation. A device reset has the effect of returning the affected device to the state it was in immediately following power-up. A device reset causes all of the device’s initialization data (e.g. routing information) to be deleted.

A device reset is invoked by the device’s receipt of a device reset message. In OCP.1, a device reset message is a special message with a unique port number and a 128-bit key (the “reset key”). The value of this key must match a previously-stored value in the device. If it does not, the device reset message has no effect.

Reset key values are set by a particular OCA command, and the memory of key values does not survive a power-up reset. If a device has not received a reset key value since the most recent power-up reset, it will ignore device reset messages. Thus, since it causes a power-up reset, a device reset message will cause the affected device to forget its reset key.

Device resets may be sent via unicast or, where supported by the control network in use, multicast.

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

11. Firmware / Software Upgrade

For devices with sufficient processing capacity, OCA supports reliable firmware/software upgrading over the network.

OCA requires that network upgrading be implemented in a way which ensures the device can recover from a failed firmware installation. This implies that 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.

Security of the firmware/software upgrade is handled through the normal security mechanism for control data (see section 7), since the firmware/software upgrade is triggered via OCP.

12. Construction of Objects

Fully configurable devices allow controllers to construct and delete worker and agent objects. This is accomplished using the OcaBlock methods:

- ConstructMember,
- ConstructMemberUsingFactory,
- DeleteMember.

When objects are constructed using ConstructMember, there is no factory to remember the parameters for their construction. These parameters are called Construction Parameters, and are defined in [OCC 1.1] for each class. When a controller calls ConstructMember, it can pass values for these parameters.

Most construction parameters have defaults which will be used if the controller does not specify otherwise.

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

Appendix A. Revision History

Date	Rev	Rel	Author	Comments
2009.01.13	0.16	pre-1.0	Stephan van Tienen	
2011.08.11	02	1.0	Jeff Berryman	Put into OCA format and edited to support OCF/OCC/OCP partitioning of architecture
2012.04.01	03	1.0	Jeff Berryman	Intermediate working copy
2012.04.12	04	1.0	Jeff Berryman	First draft of full OCF document
2012.04.25	05	1.0	Jeff Berryman	Various edits following group input
2012.05.03	06	1.0	Jeff Berryman	Various edits following group input
2012.05.04	07	1.0	Jeff Berryman	Various edits following group input
2012.05.15	08	1.0	Jeff Berryman	Fixed a couple of bad bookmark refs Added **OcaClassIDField** and updated **OcaClassID** to use it. Augmented description of **OcaClassID.**
2012.08.30	10	1.1	Jeff Berryman	Various small corrections
2012.11.01	11	1.1a	Jeff Berryman	More small corrections