NetSpire SDK API

Reference Manual

Document Version 74 ● Jun 5, 2017

Title	NetSpire SDK API
Reference	OA\DSR00300
Distribution	Commercial in Confidence

CONTACT

Open Access Pty Ltd

ACN 009 177 986

65 Epping Road • North Ryde • N.S.W. 2113

P.O. Box 61 • North Ryde BC • N.S.W. 1670

Phone +61 2 9978 7009 • Fax +61 2 9978 7099

Amendment History

Document Version	API Version	DATE	AUTHOR	NOTES
2.2	20	19/04/2013	AN	Initial release post document restructure. Incorporated positioning system and PIS control functionality.
2.3	22	08/05/2013	AN	Updated signalling section
2.4	23	28/05/2013	AN	Updated Audio Server, PaSource and PaSink class
2.5	24		AN	Added setDynamicConfiguration and getDynamicConfiguration
2.6	25	19/06/2013	AN	Removed PAController::setPaSinkGain(…) and PAController::setPaSourceLevel(…) as they are moved to PaSource and PaSink class respectively. Updated Dynamic Configuration Variables in Appendix. Updated Call hangup causes Added CallController::terminateCall(int callReference, int cleardownCauseCode) Added callType details Added Terminal Type details
26	26	09/08/2013	AN	Revision changed from 2.7 to 26 to clarify release version numbers. Updated Device and AudioServerObserver class Added DeviceModel class
27	27	16/08/2013	AN	Updated Revision from 20700 to 27 Updated PaSink::setGain and PaSource::setGain to include persistence parameter
28	28	21/08/2013	AN	Corrected definition of Default_Monitor_Speaker_Gain range to [-96, 0]
29	29	13/09/2013	AN	Updated section 7.1 Dynamic Configuration Variables. Added Default_Standard_PEI_Mic_Volume_dB and Default_Standard_PEI_Spk_Volume_dB. Default_Standard_PEI_Volume is now obsolete
30	30	27/09/2013	AN	
31	31	13/12/2013	AN	
32	32	20/01/2014	AN	

Document Version	API Version	DATE	AUTHOR	NOTES
33	33	27/06/2014	AS	Updated Template; Reset document version number to be synchronised with generic software package release number; Updated document reference number.
34	34	22/07/2014	AN	Code change only for bug fixes
35	34	04/08/2014	MF	Updated documentation to add additional information on classes, their usage, device states and supplementary fields, document objective, and appendix sections.
36	36		AN	
37	37	05/11/2014	AN	Added new functions for Passenger Information Server class
38	38	06/11/2014	AN	Added new functions for Passenger Information Server class. Code changes
39	39	21/11/2014	AN	Code optimisation and improvments
40	40	28/01/2015	AN	Added ability to use network interface bond for retrieving IP address
41	41	10/06/2015	MF	Added guidelines for porting from SDK 1.x revisions
42	42	26/06/2015	JZ	Removed guidelines for porting from SDK 1.x revisions. Code documentation cleanup.
43	43	08/07/2015	JZ	Added PaZone class reference.
44	44	31/7/2015	AN/MF	Restructured API classes and methods to accommodate the addition of PaZone, including introducing a new PaController::playMessage() to play directly to zones. Updated dictionary update to allow minimise downtime during update.
45	45	1/9/2015	MF	Resolved incorrect document referencing and section numbering
46	46	10/9/2015	AN	Added support for user recorded messages, new event types for zone creation and deletion
47	47	22/9/2015	AN	Resolved an issue causing intermittent crashes for Java applications using API via JNI
48	48	23/9/2015	AN	Resolved an issue with incorrect PaZone method name
49	49	24/9/2015	AN	Added method AudioServer::setDeviceList
50	50	29/9/2015	AN	Resolved an issue with uploading dictionary

Document Version	API Version	DATE	AUTHOR	NOTES
51	51	29/9/2015	AN	Added support for system level coordinated device health test
52	52	30/9/2015	AN	Added new device type TRAIN_RADIO that can be sent to setDeviceList()
53	53	01/10/2015	MF	Now allowing use of user provided device lists(set by setDeviceList()) in device health test
54	54	01/10/2015	MF	Resolved an issue with setDeviceList()
55	55	08/10/2015	MF	Minor updates to ensure compatibility for all supported platforms
56	56	09/10/2015	MF	Updated PAController::playMessage and the Message class to include TTS speaker parameters
57	57	13/10/2015	AN	Resolved an issue with propagating gain adjustment when using playMessage. Device supplementary fields now return zone ID strings instead of enumerated type.
58	58	23/10/2015	AN	Resolved an issue preventing test only messages to be requested through the playMessage method.
59	59	04/11/2015	AN	Resolved an issue preventing correct state of DIN to be reported.
60	60	05/11/2015	AN	Resolved an issue which was preventing correct device states from be reported in a failover mode.
61	61	15/1/2016	AN	Resolved an issue with processing cab speaker mute/unmute commands. The method PAController:: cancelMessage(unsigned int) updated to only stop DVA type messages.
62	62	01/02/2016	AN	Reporting devices with dictionary support in COMMS_FAULT state until their dictionary revision is received.
63	63	23/02/2016	AN	Resolved an issue that could result in the call to updateDictionary to fail after an aborted update.
64	64	17/03/2016	AN	Resolved an issue where MessageId supplementary field was included with a 0 content for PA announcements. Resolved an issue with playmessage and cancelmessage method variants that accept Message class arguments.
65	65	22/04/2016	AN	Increased max message length limit when communicating with Audio Servers using UDP
66	66	23/05/2016	AN	Added new methods for retrieving TTS generated announcement content

Document Version	API Version	DATE	AUTHOR	NOTES
67	67	26/08/2016	MF	Updated the Media Library update documentation.
68	68	09/09/2016	AN	Updated display update and scheduling methods
69	69	03/01/2017	MF	Updated the Passenger Information Server documentation
70	70	10/01/2017	AN	Resolved an issue related with Supplementary Fields during TR calls.
71	71	17/01/2017	AN	Updated AudioServer documentation to include Alarm class, updated ScheduleDefinition to support synchronised audio/visual message schedules.
72	72	5/06/2017	MF	Updated the Passenger Information Server documentation

Contents

FIGURES

Tables

1 Glossary

Term	Meaning
AGC	Automatic Gain Control
ANS	Ambient Noise Sensor
AoIP	Audio over Internet Protocol
CI	NetSpire Crew Intercom
CP	NetSpire Crew Panel
CXS	NetSpire Communications Exchange Server
DVA	Digital Voice Announcement
HMI	Human Machine Interface
HP	Help Point
GPS	Global Positioning System
GUI	Graphical User Interface
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
NTP	Network Time Protocol
OA	Open Access Pty Ltd.
PA	Public Address
PEI	Passenger Emergency Intercom.
PID	Passenger Information Display
QoS	Quality of Service
SDK	Software Development Kit
VoIP	Voice over Internet Protocol

Table 1 - Glossary

2 Introduction

2.1 About The NetSpire SDK Interface Library

This document describes the Open Access NetSpire Software Development Interface. The NetSpire API provides a mechanism for controlling, managing and monitoring the components and devices in a distributed NetSpire communication system. The scope of the SDK covers public address, telephony, digital voice announcement, display management, signalling and passenger information.

For more details on the NetSpire communication system, including example system topologies, and supported functionality refer to section *2.3 References*.

The NetSpire API is a cross platform library with bindings available for multiple programming languages. Network data transport and interprocess communication is handled transparently by the interface and abstracted to allow support for a different underlying communications transports.

This reference manual describes the general usage of the API and provides a detailed reference covering the functions available through the API.

2.2 About This Manual

This manual is targeted at software developers who are interfacing with the Open Access NetSpire communication system using the NetSpire SDK.

The document defines terms where required, however additional definitions and background information is available in the related documentation and should be consulted where appropriate.

2.3 References

#	DOCUMENT NUMBER	TITLE
1	OA/DSR00100	NetSpire System Alarms Reference
2	OADSR00200	NetSpire OPC-DA Reference
3	OA/DSR00400	NetSpire SNMP Reference
4	OADSR00500	NetSpire Web Administration Reference
5	OADPC00100	NetSpire Product Catalogue

Table 2 - References

2.4 Developer Support

Open Access can be engaged to provide support and software development assistance.

For any technical or sales questions, please contact Open Access using one of the following contact options listed in the table below:.

Open Access Mail Address	PO Box 61, North Ryde, NSW Australia 1670
Developer Support	+61 2 9978 7009
FAX	+61 2 9978 7099
Email	**techsupport@oa.com.au**
Web Address	http://www.oa.com.au

Table 3 - Support Contacts

To assist us with your enquiry we will need as much information as possible about the request. Please ensure you have the following information available prior to contacting Open Access for support

- ▶ Operating System, release and or distribution.
- ▶ Version of system libraries installed such as runtime library versions
- ▶ Hardware details including version numbers and configuration.
- ▶ Details of network topology and addressing, including DNS settings.
- ▶ Clear description of what you are trying to achieve and what problem you are experiencing.

2.5 NetSpire SDK Purpose and Scope

The NetSpire SDK supports controlling, configuring and monitoring NetSpire devices. Support is provided for the following functional areas:

- ▶ Controlling public address announcements locally and via network connected locations.
- ▶ Controlling telephony and intercom communication
- ▶ Controlling visual displays
- ▶ Controlling recorded audio announcements (DVA)
- ▶ Managing system configuration and setting
- ▶ Real-time monitoring of system and device status
- ▶ Real-time monitoring of faults and alarms

Supplementary classes are also provided for managing passenger information and signaling interfaces.

The NetSpire SDK API interface provides sufficient monitoring and control functions to implement a comprehensive Human Machine Interface (HMI) for control of the NetSpire communication system.

2.6 NetSpire SDK API Package Contents

The API package contains the following items:

- ▶ A shared object library file compiled for 32bit x86 GLIBC based Linux Operating Systems for programming in C++ and Java.
- ▶ A dynamic link library file compiled for Microsoft Windows Operating Systems for programming in C++, Java or C#.
- ▶ Java wrappers for supporting Java based development.
- ▶ NET wrappers for supporting Microsoft .NET development.
- ▶ Example test application source code in C++, Java and C#

3 NetSpire System

The NetSpire communication system is a distributed communication platform that supports live Public Address (PA), General and Emergency Telephony, Visual Display Control, Recorded Announcement Playback, A (DVA), Audio Recording and associated functions.,

The NetSpire communication system may be used in a range of environments including, public infrastructure, industrial, education, mining and transport. Transport applications often incorporate a wide variety of devices, functionality and communication mechanisms and for this reason, examples provided in this document will often use transport systems to illustrate usage of the SDK.

The following diagram shows a NetSpire system topology as deployed in a combined fixed infrastructure and rolling stock transport environment. In this environment all audio communication is via IP, with telephony utilising SIP and Public Address RTP or Dante™ protocols.

In the system below, the two central communication exchange servers (CXS) function as an interfacing point and provide prioritisation, co-ordination, centralised configuration as well as acting as a VoIP telephony switch. The customer application server (shown in the blue box) communicates with the NetSpire CXS servers via the NetSpire API.

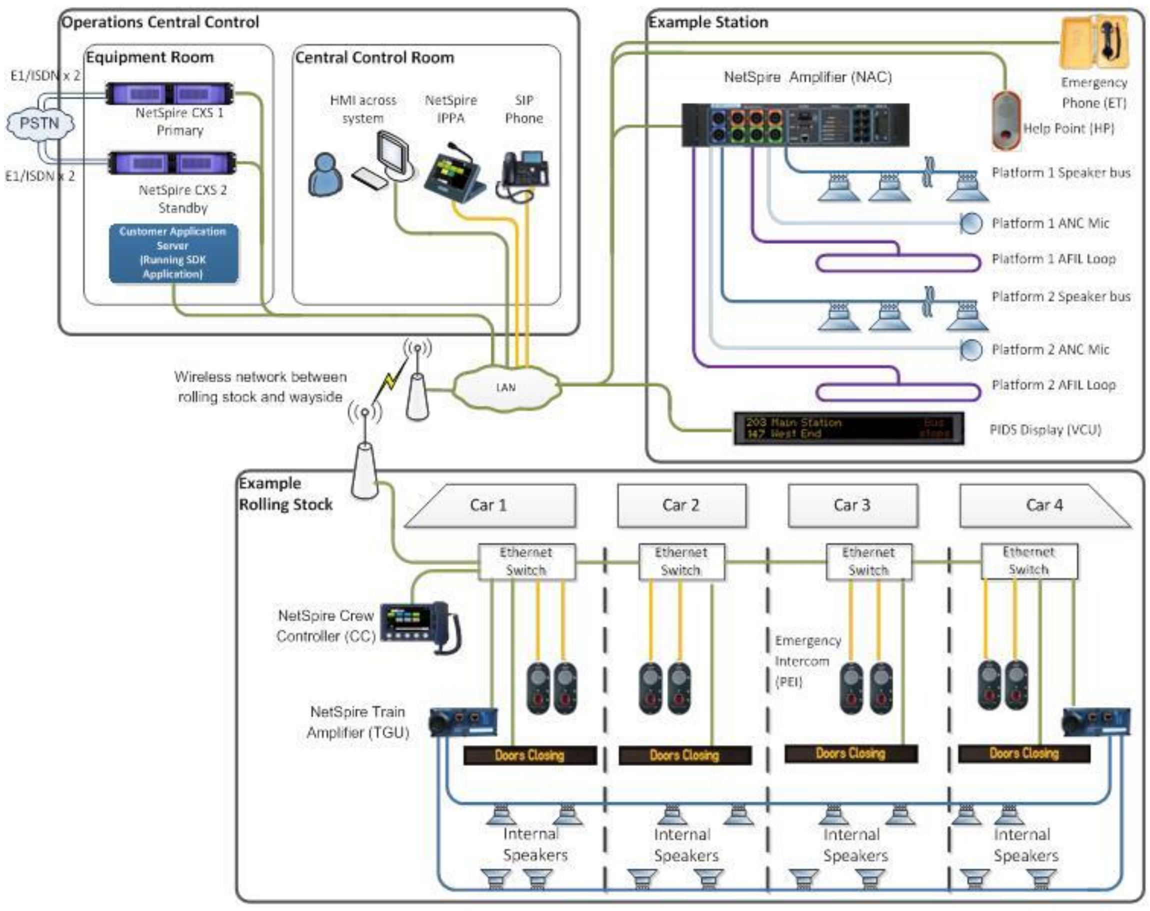


Figure 1 - Example NetSpire System Topology

3.1 NetSpire System Interfacing

Multiple interfacing methods in addition to the NetSpire SDK API are supported for interfacing into the NetSpire system. These interfacing methods include:

- **NetSpire SDK API (as covered in this document)**
  provides a library with support for C++/Java/C#/.NET to integrate with the system integrators application. Support is provided for Microsoft Windows & Linux. This is the base interface supplied with NetSpire systems.
- **Condition Monitor Logic Editor**
  Provides a graphical flowcharting tool for developing custom application and conditional logic. (see Figure 2 below).
- **NetSpire Web Services SOAP Interface**
  Provides a SOAP compliant Web Services interface into the NetSpire system.
- **OPC for SCADA**
  Provides an OPC-DA Server interface for control and monitoring. This interface provides a subset of functions as typically require in a SCADA environment.
- **SNMP**
  Provides a SNMPv2 Agent on NetSpire Servers and Devices for monitoring device status and basic functional control.

This document only covers the NetSpire SDK API interface. For information on alternate interfaces, please contact Open Access.

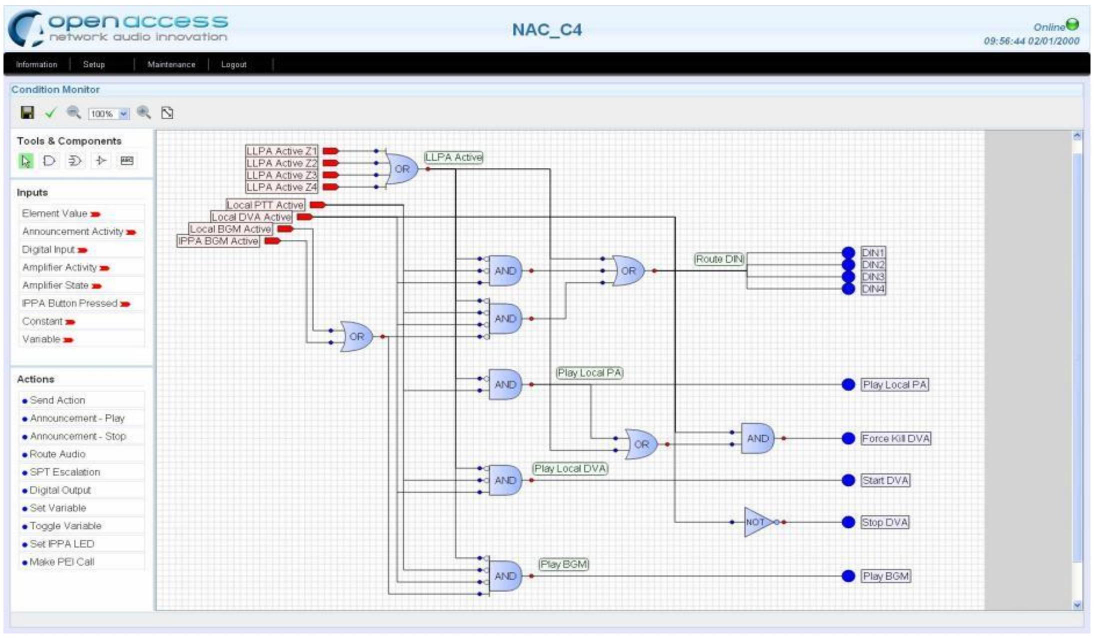


Figure 2 - NetSpire Condition Monitor Logic Editor Screenshot

3.2 NetSpire System Interfacing Topology

The manner in which external systems communicate with the NetSpire System varies in accordance with the method of interfacing used.

3.2.1 NetSpire API with Centralised Application Server

In environments where there is a centralised application server used to communication with the NetSpire system using the API (as per the system topology example shown in Figure 1) the interfacing arrangement is as per the following structure.

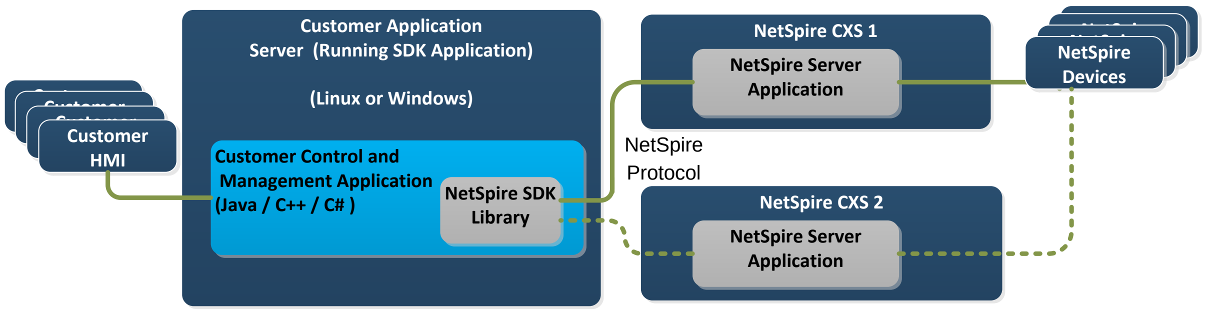

Customer Application Server (Running SDK Application)
(Linux or Windows)
Customer Control and Management Application (Java / C++ / C#)
NetSpire SDK Library
Customer HMI
NetSpire CXS 1
NetSpire Server Application
NetSpire Protocol
NetSpire CXS 2
NetSpire Server Application
NetSpire Devices

Figure 3 - Interfacing to a Customer Application Server via NetSpire API.

The customer or system integrator application links with the NetSpire API Library. The NetSpire library manages communication with the CXS servers, including selection of the correct server to use where a redundant configuration is used. In the diagram, secondary communication links are shown as dotted lines.

3.2.2 NetSpire API with direct communication from HMI Workstations

For environments, where the operator or user interface does not use or interact with a central application server, the interfacing arrangement is as per Figure 4.

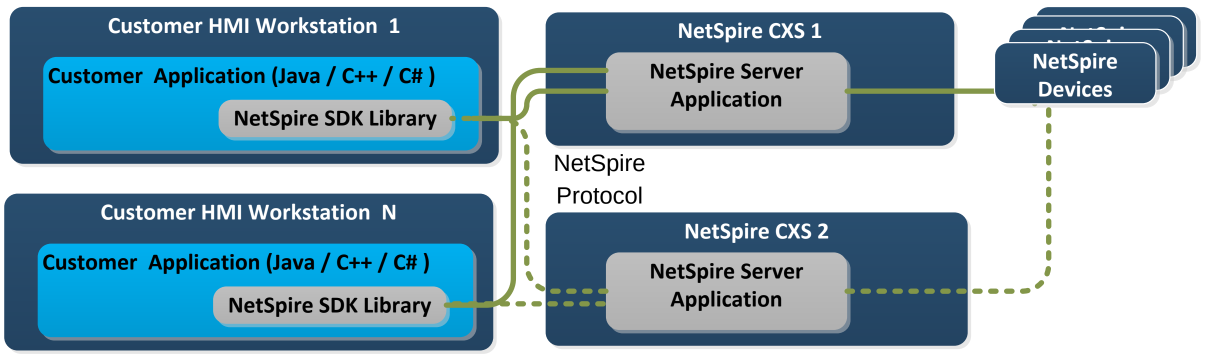

Customer HMI Workstation 1
Customer Application (Java / C++ / C#)
NetSpire SDK Library
Customer HMI Workstation N
Customer Application (Java / C++ / C#)
NetSpire SDK Library
NetSpire CXS 1
NetSpire Server Application
NetSpire Protocol
NetSpire CXS 2
NetSpire Server Application
NetSpire Devices

Figure 4 - Interfacing to a Distributed HMI applications via NetSpire API.

In this configuration, each operator workstation has a connection to the CXS. The number of connections the CXS can support will vary in accordance with the application and system size, although in most environments this is not a practical limit.

3.2.3 NetSpire Interfacing via Web Services Interface

This document covers the NetSpire API interface, however as indicated in section 3.1, other interfaces are also available. The following diagram provides an indication of the structure used with the other interfaces supported in the NetSpire System.

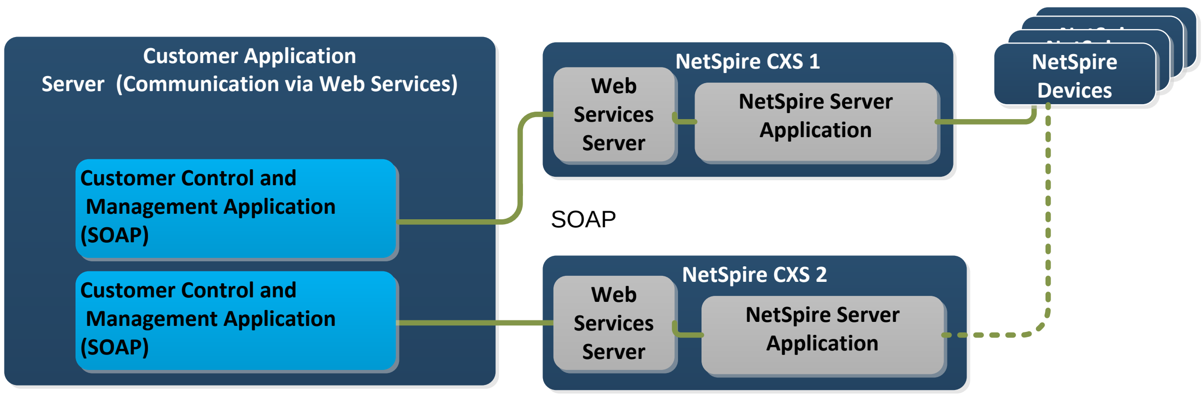

Customer Application Server (Communication via Web Services)
Customer Control and Management Application (SOAP)
Customer Control and Management Application (SOAP)
NetSpire CXS 1
Web Services Server
NetSpire Server Application
SOAP
NetSpire CXS 2
Web Services Server
NetSpire Server Application
NetSpire Devices

Figure 5 - Interfacing to NetSpire via Web Services SOAP Interface.

When interfacing to the NetSpire system via Web Services, the SOAP interface is supported at the CXS level and interaction with downstream devices is handled in the same way as it is in the other examples provided. However unlike the NetSpire API example, it is necessary for the interfacing application to communicate with both servers in redundancy supporting systems.

3.2.4 NetSpire Interfacing via OPC-DA Interface

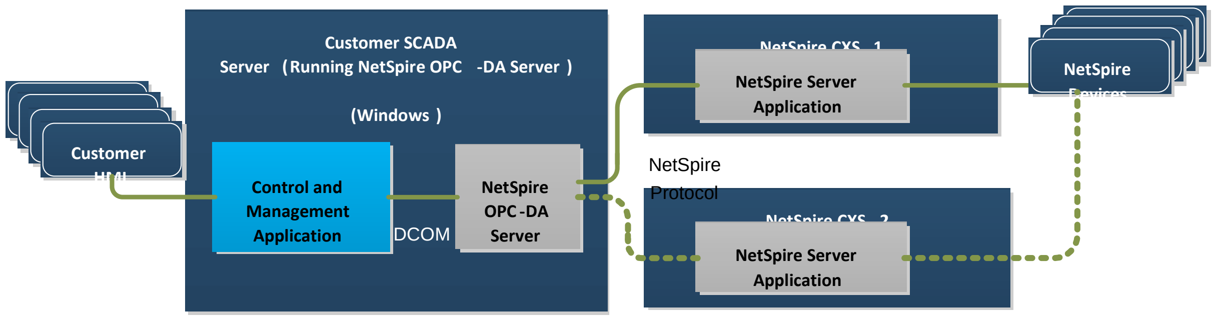

Customer SCADA Server (Running NetSpire OPC -DA Server)
(Windows)
Customer HMI
Control and Management Application
DCOM
NetSpire OPC-DA Server
NetSpire CXS 1
NetSpire Server Application
NetSpire Protocol
NetSpire CXS 2
NetSpire Server Application
NetSpire Devices

Figure 6 - Interfacing to NetSpire via OPC SCADA Interface.

The NetSpire system also supported interfacing via OPC-DA and exposes an interface as an OPC-DA Server to the OPC-DA client in the Customer SCADA system. The OPC-DA interface is currently only supported in Microsoft Windows Environments.

3.2.5 NetSpire Interfacing via SNMP Interface

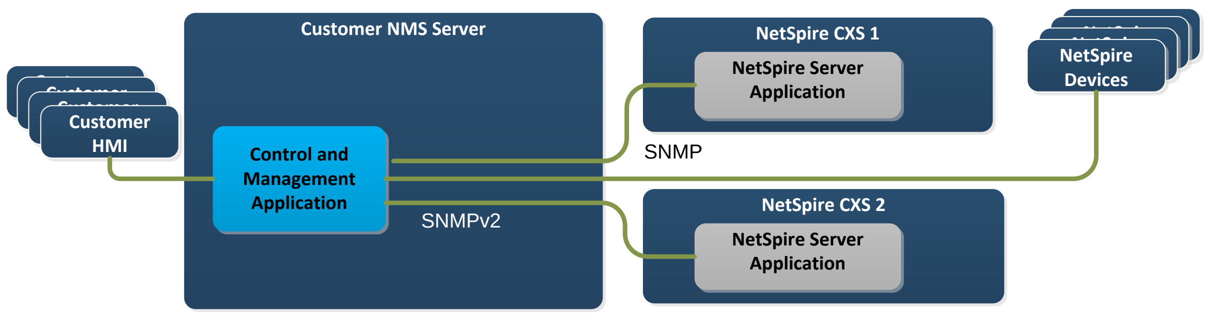

Customer NMS Server
Customer HMI
Control and Management Application
SNMPv2
NetSpire CXS 1
NetSpire Server Application
SNMP
NetSpire CXS 2
NetSpire Server Application
NetSpire Devices

Figure 7 - Interfacing to NetSpire via SNMP.

SNMP is supported on all devices and when interfacing via SNMP, it is necessary to form a connection with each of the NetSpire Devices directly. SNMPv2 is currently supported.

3.3 NetSpire System Redundancy

NetSpire systems can be deployed with or without redundant CXS servers. Where redundant communication exchange servers are used, they operate in a Master Slave arrangement where the Master / Active server is automatically elected from the available servers.

When interfacing to the system via the NetSpire API, the NetSpire Library automatically determines the correct server to use and will manage the server interaction without external direction.

In transport environments, there may also be communication servers on vehicles (Train Communication Exchanges – TCX and MCX). In this configuration, the Fixed Infrastructure and Mobile Communication servers operate in a hierarchy where the MCX/TCX devices will manage the devices local to the vehicle and when interacting with fixed infrastructure automatically determine which CXS to use.

4 NetSpire SDK Class Overview

4.1 General Classes

Class	Description
AudioServer	Provides primary communications interface to a NetSpire system
AudioServerObserver	Provides an interface for event based notification for general events
Device	Provides device status and capability information, device specific controls
DeviceModel	Provides information on device capabilities and attributes
Gain	Parameter used during setting and retrieving gain levels on audio zones
Message	Represents message to play on audio and visual devices
MessagePriority	Represents the priority level of a Message
PAController	Provides an interface to control and monitor live Public Address subsystem
PaSource	Represents an audio source
PaSink	Represents a target for PA and DVA
PaZone	PaZone class represents an audio zone, and is a collection of PaSinks
PaTrigger	Software or hardware trigger that indicates the PaSource is now activated
PaSelector	Software or hardware zone selector switch
PAControllerObserver	Provides an interface for event based notification for PA related events
CallController	Provides an interface to initiate new calls and manage existing calls
CallInfo	Provides information on an individual call
Terminalinfo	An entity that has been involved in a call
SIPTrunk	Provides information on a SIP connection to an external phone system.
ISDNTrunk	Provides information on an ISDN connection to an external phone system.
CallControllerObserver	Provides an interface for event based notification for call related events
ScheduleDefinition	Represents a schedule in the system
Alarm	Represents an alarm in the system

Table 4 - General Class Overview

4.2 Passenger Information Related Classes

Class	Description
PassengerInformationServer	Provides an interface to manage Passenger Information subsystem
PassengerInformationObserver	Provides an event based notification interface for PI related events
Vehicle	Represents a vehicle in the system such as rolling stock, busses and ferries
Station	Represents a location at which a vehicle stops during a Trip (journey)
Trip	Journey/stopping pattern: an ordered list of TripStops that a vehicle stops
PlatformInfo	Represents a location within an station
Service	A trip instance that runs at a given date and time.
TripStop	A platform at which a Vehicle stops during a Trip
ServiceStop	A TripStop with scheduled information related to a particular Service.
Line	A list of stations that are normally connected in a vehicle service
Priority	Classifies the type of Service e.g. Express vs. All Stops
NamedId	Base class for objects that require an unique id, name and abbreviation

Table 5 - Passenger Information Class Overview

5 NetSpire SDK Member Function Overview

5.1 General Classes

5.1.1 AudioServer Class

Member Function	Description	SDK Version Support
connect	Connects to the system and initialise system configuration	1
disconnect	Disconnects from the NetSpire system	1
isAudioConnected	Determines if connection to an Active server is successfully established	1
isCommsEstablished	Determines if connection to a device is successfully established	15
setDynamicConfiguration	Sets system wide configuration variables	19
getDynamicConfiguration	Retrieves current value of system wide configuration variables	19
registerObserver	Registers an observer object for event based notification	12
getPAController	Get PAController instance providing interface to the PA subsystem	16
getCallController	Get CallController instance providing interface to the Call subsystem	18
initiateDeviceHealthTest	Initiates health test for the specified devices	1
setDeviceIsolation	Isolate (disable)/deisolate (enable) a device	1
setDeviceList	Sets the list of devices that will be returned by getDeviceStates	49
getDeviceStates	Queries states of all the devices on the NetSpire system	1
updateDictionary	Updates the contents of media dictionary	1
getDictionaryChangeset	Returns the current version history of media dictionary	18
getDictionaryItems	Returns the content of the current media dictionary	18
setOutputGain	Set default output gain on all amplifiers	1
getOutputGain	Query default output gain	1
getRole	Returns the current escalation level of the device	25
uploadFile	Allows a file to be uploaded to the connected server	25
getSystemRevision	Returns version number for the entire system	19
isSystemRevisionConsistent	Indicates if all the devices within the system have the correct revision	19
getSDKRevision	Returns the current version of the Netspire SDK	19
initiateShutdown	Initiates shutdown process on the NetSpire system for graceful shutdown	1
getEventList	Returns a list of all unread events raised and stored in the connected Communications Exchange Server.	17
getLogList	*Returns the list of current log items stored in the connected server*	*Refer to 5.3*
addLogItem	*Append a new log item to the log items stored in the connected server*	*Refer to 5.3*
enableAlarmManagement	*Enables alarm managements and retrieves all alarms from active server*	*71*
disableAlarmManagement	*Disables alarm management.*	*71*
getAlarms	*Retrieves all alarms from active server*	*71*

Table 6 - AudioServer Class Member Function Overview

Obsoleted Member Function	Description	Recommended Replacement Method
getAudioSinks	Returns a list of all the audio sinks (audio zones) defined in the system.	PAController::getPaSinks. Replacement method has identical arguments and return value.
playMessage	Initiate message on Audio Sinks and/or Visual Displays	PAController::playMessage. Replacement method requires zone list and not sink list in arguments.
playMessage	Initiate message on Audio Zones and/or Visual Displays	PAController::playMessage. Replacement method has identical arguments and return value.
cancelMessage	Cancel previously initiated message	PAController::cancelMessage. Replacement method has identical arguments and return value.
createSchedule	Schedule messages on Audio Zones and or Visual Displays	PAController::createSchedule. Replacement method has identical arguments and return value.
listSchedules	List all scheduled messages	PAController::listSchedules. Replacement method has identical arguments and return value.
deleteSchedule	Delete a scheduled message	PAController::deleteSchedule. Replacement method has identical arguments and return value.

Table 7 - AudioServer Class – Obsoleted Member Functions

5.1.2 AudioServerObserver Class

Member Function	Description	SDK Version Support
onCommsLinkUp	Callback function - called when connection to a device is established	15
onCommsLinkDown	Callback function - called when connection to a device is lost	15
onAudioConnected	Callback function indicating connection to an active server is established	14
onAudioDisconnected	Callback function indicating connection to an active server is lost	14
onDeviceStateChange	Callback indicating a Device state is updated	14
onDeviceDelete	Callback indicating a Device state is removed	14
onVUMeterUpdate	Callback indicating VU meter is updated during active PA	15
onConfigUpdate	Callback indicating a configuration value is updated	15
onStateUpdate	Callback function indicating the state of connected device value is updated	15
onRealTimeItemUpdate	Callback indicating a real-time configuration item is updated	15
onDebugMessage	Callback function indicating the SDK library published a debug message	15

Table 8 - AudioServerObserver Class Member Function Overview

5.1.3 Device Class

Member Function	Description	SDK Version Support
getName	Returns the name of the device.	1
getState	Returns the current state of this device	1
getStateText	Returns the current state of this device as string	1
getSupplementaryFields	Returns the supplementary fields of this device state	1
getHealthTestStatus	Query the status of the most recent health test requested	1
initiateDeviceHealthTest	Initiates health test on a device	1
getDictionarySupport	Queries if the device supports media dictionary	5
getDictionaryVersion	Query the version number of the dictionary maintained by device	1
getDictionaryUpdateStatus	Query the status of the most recent dictionary update on device	1
getInputState	Returns status of externally set input (DIO) on device	1
getOutputState	Returns status of digital output (DIO) on device	1
setOutputState	Sets status of digital output (DIO) on device	1
getDstNo	Returns device destination number	26
getPortNo	Returns device port number	26
getIP	Returns device IPv4 address	26
getSoftwareRevision	Retrieves the version number of the installed firmware	5
getLocationName	Retrieves the location name that is set when the device is configured	15
getLocationId	Retrieves the location ID that is set when the device is configured	26
getDeviceIndex	Retrieves the device index that is set when the device is configured	26
getDeviceClass	Returns the class that the device belongs to	26
getDeviceModel	Returns the device model for the device	26
setCondition	Sets the value of a boolean condition variable	42
setCondition	Sets the value of a string condition variable	44
getCondition	Gets the value of a condition variable	*Refer to 5.3*
initiateRestart	*Sends a restart request to the device*	*Refer to 5.3*

Table 9 - Device Class Member Function Overview

5.1.4 PAController Class

Member Function	**Description**	**SDK Version Support**
getPaSources	Returns the list of available PA sources accessible to the user	16
getPaSinks	Returns all audio sinks that can make announcements from a specifed source	16
getPaSinks	Returns a list of all audio sinks	42
getDisplays	Returns a list of all Visual Displays	44
getPaZones	Returns a list of all audio zones	42
setPaMonitorZone	*Allows an application to monitor a particular sink on another sink*	*Refer to 5.3*
getHwPaTriggers	Returns a list of available hardware triggers for a particular source.	16
getHwPaTriggerState	Returns current state of a hardware trigger	16
createSwPaTrigger	Creates a software trigger associated with the source	16
deleteSwPaTrigger	Removes a software trigger	16
activateSwPaTrigger	Activates a software trigger, which starts announcement from source	16
deactivateSwPaTrigger	Deactivates a software trigger, stopping the announcement	16
getPaSelectors	*Returns a list of all available zone selectors for the source*	*Refer to 5.3*
addPaZoneToSelector	*Associates a new zone with a selector*	*Refer to 5.3*
deletePaZoneFrom Selector	*Removes a zone from selector*	*Refer to 5.3*
getPaZonesForSelector	*Returns zones associated with a zone selector*	*Refer to 5.3*
enablePaSelector	*Activates a zone selector*	*Refer to 5.3*
disablePaSelector	*Deactivates a zone selector*	*Refer to 5.3*
getPaSelectorState	*Returns current state of a zone selector*	*Refer to 5.3*
registerObserver	Registers an observer object for PA event notifications	16
playMessage	Initiate message on Audio Zones and/or Visual Displays	44
playMessage	Extended form for initiating message with TTS and Template support	64
cancelMessage	Cancel previously initiated message	44
cancelMessage	Cancel previously initiated message	64
cancelMessageBy PaZone	*Cancel playing or queued message(s) on zones*	*Refer to 5.3*
createSchedule	Schedule messages on Audio Zones and or Visual Displays	44
listSchedules	List all scheduled messages	44
deleteSchedule	Delete a scheduled message	44

Table 10 - PAController Class Member Function Overview

Obsoleted Member Function	Description	Recommended Replacement Method
attachPaSource	Allows an application to bind to and manage a source	Method is no longer required and can be removed without a side effect.
detachPaSource	Allows an application to unbind from a source	Method is no longer required and can be removed without a side effect.
getPaSourceState	Returns the current state of a PaSource	PaSource::getHealthState() can now be used to retrieve PaSource state.
getAttachedPaSinks	Returns a list of all audio sinks that are currently attached to the source	PASource::deleteSchedule. Replacement method has identical arguments and return value.
attachPaSink	Manually attaches a sink to establish a route for audio from source	PaSource::attachPaZone can now be used to establish a route from an audio source to a list of zones.
detachPaSink	Detaches from sink; removing the route from source to sink	PaSource::attachPaZone can now be used to remove a route from an audio source to a list of zones.
getPaSinkState	Retrieves current state of a sink	PaSink::getActivityState() can now be used to query the current activity state of a sink,

Table 11 - PAController Class – Obsoleted Member Functions

5.1.5 DeviceModel Class

Member Function	Description	SDK Version Support
getName	Returns the model of the device	26
getDigitalInputCount	Returns number of digital inputs on device	26
getDigitalOutputCount	Returns number of digital outputs on device	26
getAudioOutput Channels	Returns number of audio output channels on device	26

Table 12 - DeviceModel Class Member Function Overview

5.1.6 Gain Class

Member Function	Description	SDK Version Support
setLevel	Sets gain level	1
getLevel	Retrieves selected gain level	1

Table 13 - Gain Class Member Function Overview

5.1.7 Message Class

Member Function	Description	SDK Version Support
setRequestID	Provides an ID for the playback request	56
getRequestID	Returns the ID for the playback request	56
setPriority	Specifies the requested message priority	56
getPriority	Returns the message priority	56
setPreChime	Specifies the chime played prior to message	56
getPreChime	Returns the chime played prior to message	56
setPostChime	*Specifies the chime played after message*	*Refer to 5.3*
getPostChime	Returns the chime played after message	56
setGain	Specifies gain offset applied during playback	56
setAudioMessageType	Sets audio message type to one of the supported formats	56
getAudioMessageType	Returns audio message type.	56

setVisualMessageType	Sets visual message type to one of the supported formats	56
getVisualMessgeType	Returns visual message type	56
setAudioMessage	Sets audio message contents to a list of dictionary items	56
getAudioMessage	Returns list of audio message dictionary items	56
setAudioMessage	Sets audio message contents to TTS generated text	56
getAudioMessageTTSText	Returns the text set for TTS playback	56
getAudioMessageTTSLanguage	Returns the language set for TTS playback	56
getAudioMessageTTSVoice	Returns the voice set for TTS playback	56
getAudioMessageTTSEncoding	Returns the encoding set for TTS text	56
setAudioMessage	*Sets audio message contents to template*	*Refer to 5.3*
setVisualMessage	Sets visual message contents to a list of dictionary items	56
getVisualMessageDictionaryItems	Returns list of visual message dictionary itmes	56
setVisualMessage	Sets visual message contents to TTS generated text	56
getVisualMessageText	Returns the text set for visual display	56
setVisualMessage	*Sets visual message contents to template*	*Refer to 5.3*
retrieveAudioMessage	Retrieves audio message contents	66

Table 14 - Message Class Member Function Overview

5.1.8 PaSource Class

Member Function	Description	SDK Version Support
setGain	Sets gain level for audio source	25
getGain	Returns gain level for audio source	25
getHealthTestMode	*Returns health test mode for a PaSource*	*Refer to 5.3*
initiateHealthTest	*Initiates health test for a PaSource*	*Refer to 5.3*
getHealthState	*Returns the status of a PaSource*	*Refer to 5.3*
getAttachedPaZones	Returns a list of all audio zones that are currently attached to the source	44
attachPaZone	Manually attaches a zone to establish a route for audio from source	44
detachPaZone	Detaches from zone; removing the route from source to zone	44
detachAllPaZones	Detaches all zones; removing the route from source to all attached zones	44
getAttachState	Retrieves the attachment status of the Pa Source.	44

Table 15 - PaSource Class Member Function Overview

5.1.9 PaSink Class

Member Function	Description	SDK Version Support
setGain	Sets gain level for members of audio sink	25
getGain	Returns gain level for members of audio sink	25
setPAMonitoring	Enables/disables PA and DVA monitoring	43
getPAMonitoring	Returns current PA and DVA monitoring state	43
mute	*Mute audio sink*	*Refer to 5.3*
unmute	*Unmute audio sink*	*Refer to 5.3*
isMuted	*Returns true if sink is muted*	*Refer to 5.3*
getHealthTestMode	*Returns health test mode for a PaSink*	*Refer to 5.3*

initiateHealthTest	*Initiates health test for a PaSink*	*Refer to 5.3*
getHealthState	*Returns the current health test state for a PaSink*	*Refer to 5.3*
setCallMonitoring	*Enables/disables call monitoring*	*Refer to 5.3*
getCallMonitoring	*Returns current call monitoring state*	*Refer to 5.3*
getZones	Returns the PaZones this PaSink is a member of	44

Table 16 - PaSink Class Member Function Overview

5.1.10 VisualDisplay Class

Member Function	Description	SDK Version Support
getName	Returns the device name assigned to the display device	44
getCommsType	Retrieves the method used for communications with the device	44
getAddress	Returns the address used for communications to the display device	44
getHealthState	Returns the current health test state for the VisualDisplay	44
getHealthTestMode	*Returns health test mode for the VisualDisplay*	*Refer to 5.3*
getHealthTestStatus	*Returns the current health test state for the display*	*Refer to 5.3*
initiateHealthTest	*Initiates health test for the VisualDisplay*	*Refer to 5.3*

Table 17 - VisualDisplay Class Member Function Overview

5.1.11 PaZone Class

Member Function	Description	SDK Version Support
getMembers	Returns the list of member PaSinks	44
setMembers	*Assigns member PaSinks comprising a PaZone*	*Refer to 5.3*
mute	*Mute all members of the audio zone*	*Refer to 5.3*
unmute	*Unmutes all members of the audio zone*	*Refer to 5.3*
isMuted	*Returns true if all members of the audio zone are muted*	*Refer to 5.3*

Table 18 - PaZone Class Member Function Overview

5.1.12 PAControllerObserver Class

Member Function	Description	SDK Version Support
onPaSourceUpdate	Callback indicating a PA Source is updated	16
onPaSourceDelete	Callback indicating a PA Source is deleted	16
onPaSinkUpdate	Callback indicating a PA Sink is updated	16
onPaSinkDelete	Callback indicating a PA Sink is deleted	16
onPaTriggerUpdate	Callback indicating a PA Trigger is updated	16
onPaTriggerDelete	Callback indicating a PA Trigger is deleted	16
onPaSelectorUpdate	Callback indicating a PA Selector is updated	16
onPaSelectorDelete	Callback indicating a PA Selector is deleted	16
onAudioMessageRetrievalComplete	Callback indicating message retrieval request status	66

Table 19 - PaControllerObserver Class Member Function Overview

5.1.13 CallController Class

Member Function	Description	SDK Version Support
createDestination	Creates a custom escalation and handling rule per extension	18
createCall	Initiates a new call between two parties	18
answerCallOnTerminal	Answers an incoming call on specified terminal	18
resumeCall	Resumes a held call l	18
resumeCallOnTerminal	Resumes a held call on the specified terminal	18
transferCall	Transfers a call party to new party	18
holdCall	Puts a call on hold	18
terminateCall	Ends a call	18
terminateCall	Ends a call with specific clear-down cause	18
getCalls	Returns a list of active calls	18
getCDRMessages	Returns a list of call detail records	18
getTerminalList	Returns a list of all defined terminals	18
getSIPTrunks	Returns a list of SIP Trunks that are configured in the system	18
getISDNTrunks	Returns a list of ISDN Trunks that are configured in the system	18
registerObserver	Registers an observer object for PA event notifications	18

Table 20 - CallControllerObserver Class Member Function Overview

5.1.14 SIPTrunk Class

Member Function	Description	SDK Version Support
getDeviceID	*Returns the device where the trunk is installed and configured*	*Refer to 5.3*
getType	*Returns the type of trunk*	*Refer to 5.3*
getType	*Returns the name assigned to the trunk*	*Refer to 5.3*
getStatus	*Returns the current status of the Trunk l*	*Refer to 5.3*

Table 21 - SIPTrunk Class Member Function Overview

5.1.15 ISDNTrunk Class

Member Function	Description	SDK Version Support
getDeviceID	*Returns the device where the trunk is installed and configured*	*Refer to 5.3*
getType	*Returns the type of trunk*	*Refer to 5.3*
getType	*Returns the name assigned to the trunk*	*Refer to 5.3*
getLayer1Status	*Returns the current Layer 1 status of the Trunk l*	*Refer to 5.3*
getLayer2_3Status	*Returns the current Layer 2 and 3 status of the Trunk l*	*Refer to 5.3*

Table 22 - ISDNTrunk Class Member Function Overview

5.1.16 CallControllerObserver Class

Member Function	Description	SDK Version Support
onCallUpdate	Callback indicating a call is created or updated	18
onCallDelete	Callback indicating a call is deleted	18
onCDRMessageUpdate	Callback indicating a Call Detail Record (CDR) is created or updated	18
onCDRMessageDelete	Callback indicating a Call Detail Record (CDR) is deleted	18
onTerminalUpdate	Callback indicating a Terminal is created or updated	18
onTerminalDelete	Callback indicating Terminal is deleted	18

Table 23 - CallControllerObserver Class Member Function Overview

5.1.17 Alarm Class

Member Function	Description	SDK Version Support
acknowledgeAlarm	Acknowledges an alarm in the system	71

5.2 Passenger Information Specific Classes

5.2.1 PassengerInformationServer Class

Member Function	Description	SDK Version Support
getServerStatus	Returns the current status of the signalling server	20
getLines	Returns the list of all the lines defined in the system	20
getServices	Returns the list of all the services defined in the system	20
getServiceStops	Returns the list of all the service stops defined in the system	38
getVehicles	Returns the list of all the vehicles defined in the system	20
getStations	Returns the list of all the stations defined in the system	20
getPlatforms	Returns the list of all the platforms defined in the system	37
getTrips	Returns the list of all trips defined in the system	37
getTripStops	Returns the list of all trip stops defined in the system	38
registerObserver	Registers an observer object for PIS event notifications	20

Table 24 - PassengerInformationServer Class Member Function Overview

5.2.2 PassengerInformationObserver Class

Member Function	Description	SDK Version Support
onServerStatusUpdated	Callback indicating status of the signalling server	20
onLineUpdated	Callback indicating a line is created or updated	20
onLineRemoved	Callback indicating a line is removed	20
onVehicleUpdated	Callback indicating a vehicle is created or updated	20
onVehicleRemoved	Callback indicating a vehicle is removed	20
onStationUpdated	Callback indicating a station is created or updated	20
onStationRemoved	Callback indicating a station is removed	20
onPlatformInfoUpdated	Callback indicating a platform is created or updated	20
onPlatformInfoRemoved	Callback indicating a platform is removed	20
onServiceUpdated	Callback indicating a service is created or updated	20
onServiceRemoved	Callback indicating a service is removed	20
onTripUpdated	Callback indicating a trip is created or updated	37
onTripRemoved	Callback indicating a trip is removed	37

5.2.3 NamedId Class

Member Function	Description	SDK Version Support
getId	Returns the numeric ID of the object	20
getName	Returns the name of the object	20
getAbrv	Returns the abbreviation of the object	20
setName	Sets the name of the objec	20
setAbrv	Sets the abbreviation of the object	20

Table 25 - PassengerInformationObserver Class Member Function Overview

5.2.4 TripStop Class

Member Function	Description	SDK Version Support
getTripId	Returns the parent Trip	20
getPlatformId	Returns the platform where vehicle stops or passes	20
isStopping	Indicates if the vehicle stops at the station or passes through.	20
getSchArvOffset	Returns the arrival time (as an offset from startTime)	20
getSchDepOffset	Returns the departure time (as an offset from startTime)	20

Table 26 - TripStop Class Member Function Overview

5.2.5 Trip Class

Member Function	Description	SDK Version Support
getLineId	Returns the Line in which the trip travels	20
getDirection	Returns the direction of the trip	20
getPriority	Returns the priority (express vs all stops) of the vehicle on the trip	20
getPriorityText	Returns the priority in textual representation	20
getStopsListAsString	Returns the stops Ids defined in the trip as comma separated string	20

Table 27 - Trip Class Member Function Overview

5.2.6 ServiceStop Class

Member Function	Description	SDK Version Support
getServiceId	Returns the parent Service	20
getTripStopId	Returns the TripStop Id from which this ServiceStop was derived	37
getServiceStartTime	Returns the time when the vehicle started the trip in the first station	20
getArvDelay	Returns the expected delay for the vehicle to arrive the station	20
getDepDelay	Returns the expected delay for the vehicle to depart the station	20
getArvTime	Returns the scheduled arrival time	20
getDepTime	Returns the scheduled departure time	20

Table 28 - ServiceStop Class Member Function Overview

5.2.7 Service Class

Member Function	Description	SDK Version Support
getTripId	Returns the Trip that formed the base of this service	20
getState	Returns the state of the service	20
getStateText	Returns the textual representation of the state of the service	20
getVehicleId	Returns the vehicle performing the service	20
getStopsSchedule	Returns a list of all stops included in the service	20
getStartTime	Returns the time vehicle has started the service	20

Table 29 - Service Class Member Function Overview

5.2.8 Vehicle Class

Member Function	Description	SDK Version Support
getNumCars	Returns the number of cars in the vehicle	20
getCurrentServiceId	Returns the service Id that the vehicle is currently serving	20
getCurrentService	Returns the service object that the vehicle is currently serving	20
getServices	Returns the list of all services scheduled to be served by this vehicle	20
getDirection	Returns the direction in which vehicle moves	20
getCurrentLocation	Returns the current location of the vehicle	20
getState	Returns the state of the vehicle	20
getStateText	Returns the textual representation of the state of the vehicle	20
getOpeningDoors	Returns the side that will open at next/current stop	20
getOpeningDoorsText	Returns the textual representation of the side that will open at next/current stop	20
getOpenedDoors	Returns the side that is currently open	20
getOpenedDoorsText	Returns the textual representation of the side that is currently open	20
createService	Creates a new service for a vehicle	37
deleteService	Deletes a service from the system	37
clearServices	Removes all services from the system	37
replaceService	Removes all existing services from the system and creates a new service	37

Table 30 - Vehicle Class Member Function Overview

5.2.9 PlatformInfo Class

Member Function	Description	SDK Version Support
getStationId	Allows an application to get the unique station id where this platform reside in	20
getLocationId	Returns the location which represents where the platform is in the network position system	20
getState	Returns the state of the platform	20
getPassers	Returns a list of all service-stops including this platform in scheduled services	20

Table 31 - PlatformInfo Class Member Function Overview

5.2.10 Station Class

Member Function	Description	SDK Version Support
isMajor	Returns if the station is defined as a major station	20
getListOfPlatforms	Returns list of platforms within this station	20
getPlatform	Returns information on selected platform	20

Table 32 - Station Class Member Function Overview

5.3 Extended and Preliminary Function Support

Functions that are not provided with a specific SDK Version support number are not formally supported as at the release date of this document, however the function may be available, but subject to change before formal support is provided. For clarification relating to the availability of funcitons that are referred to this seciton please contact Open Access.

6 NetSpire API I/O Multiplexing

The SDK supports operating in synchronous (polled) or asynchronous mode (event driven) mode as preferred by the integrator. To support asynchronous operation, the library is threaded and maintains an independent event loop in a dedicated for handling asynchronous event notification. To operate in asynchronous mode, applications must create an observer for the desired events. If an observer is not created, events are processed

7 NetSpire SDK Common Data Types

This section documents the common data type definition used by the NetSpire SDK API. Common data types used in the API include:

- ▶ Device IDs
- ▶ Device State Supplementary Fields
- ▶ Dynamic Configuration Variables
- ▶ Dictionary Changeset Definition File

These types are used as arguments or return types in various API functions. The following sections cover each of the data types used in by the API.

7.1 NetSpire Device Abbreviations

This document refers to devices by category of functionality; called a “Device Type” as well as by abbreviations for specific devices. The following table provides an overview of NetSpire Devices, the functionality provided as well as the abbreviation used to refer to the devices. For more information on specific devices, refer to NetSpire Product Catalogue.

Device Name	Description	Abbreviation
Communications Exchange	Centralized coordination and control server.	CXS
Train Communication Exchance	Communications Exchange for rolling stock environment.	TCX
Network Audio Controller	Multi-channel networked amplifier	NAC
Network Amplifier Module	Compact multi-channel networked amplifier	NAM
Modular Control Server	Modular communication gateway and controller providing integraed functionality in addition to operating as a Communications Exchange	MCX
Train Gateway Unit	Rolling stock multi-channel networked amplifier	TGU
Train Radio	Rolling stock wireless communications unit	TR
Network Audio Router	Multi-channel networked audio router	NAR
Passenger Emergency Intercom	Rolling stock based emergency intercom	PEI
Help Point	VoIP Intercom or general Help Point	HP
Emergency Telephony	VoIP Emergency Telephone with handset	ET
Crew Intercom	Driver Interface with handset for rolling stock environment.	CI
Crew Panel	Driver Interface with headset for rolling stock environment.	CP
Crew Controller	Graphical Touch Screen User Interface for rolling stock environment.	CC
Monitor Speaker	Audio Monitor Speaker	MSPK
Cabin Audio Controller	Audio gateway for interfacing with external devices in rolling stock environment	CAC
IP Paging Station	Graphical touch screen paging station and emergency telephony call management terminal	IPPA
Video Control Unit	LCD / LED Visual Display Controller	VCU

Table 33 - NetSpire Device Abbreviations

7.2 Device Types

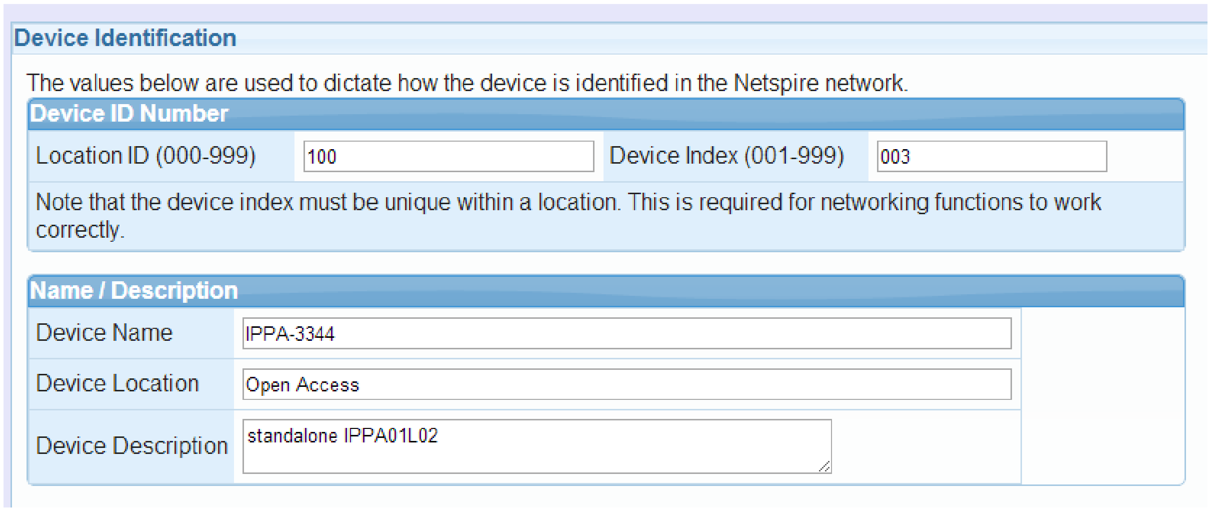
The API provides access and support for certain functions in accordance with the category of the device. The categorisation of NetSpire devices is called “Device Type” in this document and also maps directly to the public DeviceClass Enumerator (See Section8.4.5.20). The NetSpire devices associated with each device type are as per the following table.

Device Type	DeviceClass ENUM	Description	Associated Devices
Unknown	UNKNOWN_DEVICE	Unknown	-
Communications Exchange	COMMUNICATIONS_EXCHANGE	Responsible for centralized coordination and control.	CXS, TCX, MCX
Network Audio Controller	NETWORK_AUDIO_CONTROLLER	Networked audio processing devices	NAC, NAM, NAR, TGU
Operator Interface	OPERATOR_CONSOLE	Network attached devices for users to interact with.	CI, CP, CC, IPPA
Monitor Speaker	MONITOR_SPEAKER	Devices used for external interfacing to specific external interfaces or for the provision of supervisory monitoring.	MSPK, CAC
Help Point	HELP_POINT	Devices for end user two way interaction	PEI, HP, ET
Display	PASSENGER_INFORMATION_DISPLAY	LCD / LED Information Display	VCU
Train Radio	TRAIN_RADIO	Rolling stock wireless communications unit	TR

Table 34 - NetSpire Device Type Categorisation

7.3 Device IDs

The NetSpire devices that comprise a system are identified using a used specified Device ID. The Device ID is assigned to every device during system commissioning. Methods of assigning device ID's include the NetSpire Web Administration Interface accessible on each NetSpire device. In the NetSpire Web Interface, Device ID is shown as "Device Name" in the Device Identification setup screen.


Device Identification

The values below are used to dictate how the device is identified in the Netspire network.

Device ID Number

Location ID (000-999)	100	Device Index (001-999)	003

Note that the device index must be unique within a location. This is required for networking functions to work correctly.

Name / Description

Device Name	IPPA-3344
Device Location	Open Access
Device Description	standalone IPPA01L02

Figure 8 - Setting Device ID via the Web Administration Screen

The NetSpire API uses Device IDs as arguments or includes Device IDs in returned information.

Applications may determine which Device ID's are present in the system via introspection by iterating through the DeviceState Array as follows:

```
netspire.DeviceStateArray tmpDevices = gAudioServer.getDeviceStates();
for (int i = 0; i < tmpDevices.Count; i++) {
deviceID=tmpDevices.ElementAt(i).getName().ToString()
);
```

7.4 Device States and Supplementary Fields

The API provides facilities to monitor the status of all devices in the NetSpire system.

Equipment state is modelled by the concept of Device State and Supplementary Fields. These values contain information on status and methods to read this information are provided in the Device class (refer Section 8.4).

Client applications may retrieve this information synchronously or asynchronously in accordance with the following:

1- Polling: Calling the method AudioServer::getDeviceStates) returns a list of Device objects, where each Device object includes status information of a device.

2- Event based: The client application can register observers which are executed by the API library in the event of a status change. Some of the observers send updated status information by sending an updated Device object that includes the updated state.

Device State is modelled as an enumerated type that indicates the high level state of a device. Table 35 below lists all the possible states that can be reported by the API for a device.

Note some of these states are not applicable for all device types and details of state applicability are provided in the following sections. Also note the term *call* as used in the following table describes a range audio activities including single and bi-directional telephony calls, PEI, emergency intercom calls, operator to operator calls, live and recorded PA announcements.

Idle	No call (including telephony calls, PEI calls or PA announcements) in progress. Device is available to make/receive a call/PA/DVA.
Alerting	The device is processing a call request, but a call has not yet been established.
Active	A call is in progress (i.e. in any call state other than Idle).
Held	The call at the device is on hold. Only applies to telephony calls.
Escalated	A call has been escalated. Provided on Help Point devices to indicate the call is altering to an alternate location or device.
Isolated	The device is disabled and not available for operational use..Devices may be isolated manually via the API & Web Interface or automatically as a result of external interfaction. For example a digital input ma y be programmed to activate a bypass function on an Operator Interface. Devices will also report as being isolated during firmware updates and device initialisation where they are not available for operational use.
Faulty	The device is unable to function due to a fault (e.g. self test failure or detected internal failure).
Comms Fault	The device state cannot be determined due to being unreachable. This may be due to network fault or power loss.

Table 35 - Device Call States Overview

Supplementary fields include additional information on the device status in the form of key value pairs. These fields are intended to provide extended information to the client application supplementing the base Device State. Information in the supplementary fields is only relevant to the current state from the point of view of each device, such as:

- ▶ Specific fault codes for a currently fault device
- ▶ Dictionary ID for an recorded announcement that is playing if a Network Audio Controller is Active.
- ▶ Information on the originator of a call, If device is currently Alerting with an incoming call.
- ▶ Information for each party in the call If a device is Active in a two party call.
- ▶ The zones (sinks) where the PA is routed to If a Network Audio Controller device is Active making a live PA announcement.

- ▶ The source ID where the PA is initiated from If a Network Audio Controller device is Active playing a live PA announcement.

The supplementation field information is included in the Device object, and can be queried by the API method Device::getSupplementaryFields (refer Section 8.4).

The following sections define the Device States and Supplementary Fields that are published by the API for each device type.

7.4.1 Communications Exchange – Device States and Supplementary Fields

Communications Exchange Device Type includes server devices with control, coordination, supervision, prioritisation and monitoring role. The API publishes the following Device States and Supplementary Fields for devices of this type:

State	Valid	Description	Supplementary Field
Idle	Y	Communications Exchange Server is healthy and currently operating in secondary role.	Fault_Code
Isolated	Y	Communications Exchange Server has been disabled by an external controller or UI. The unit will not be considered for use as a primary server.	Fault_Code
Alerting	N	N/A	
Active	Y	Communications Exchange Server is healthy and currently operating in primary role.	Fault_Code
Hold	N	N/A	
Escalated	N	N/A	
Faulty	Y	A fault has been detected on the unit. If the unit is still capable of being used (non-fatal fault), the fault information is shown in the Supplementary Field of the Idle state (if secondary) or the Active state (if primary).	Fatal_Code
Comms Fault	Y	The primary Communications Exchange Server cannot communicate with or detect the other on the train's network. This may indicate either a network fault or failed unit.	N/A

Table 36 - Reported Communications Exchange Device Type States

Supplementary Fields	Valid Values
Fault_Code	Refer to Table 50
Fatal_Fault_Code	Refer to Table 50

Table 37 - Reported Communications Exchange Server Supplementary Fields

7.4.2 Network Audio Controller - States and Supplementary Fields

This device type includes the devices of Device Type “Network Audio Controller”. The API publishes the following Device States and Supplementary Fields for devices of this type:

State	Valid	Description	Supplementary Field
Idle	Y	DVA/PA is not playing any Audio System controlled message. The supplementary field is used to display non-fatal faults with the device.	Fault_Code
Isolated	Y	DVA/PA has been disabled by an external controller or UI. Requests to play messages will or stream PA will be ignored.	Fault_Code
Alerting	N	N/A	
Active	Y	DVA/PA is playing an Audio System controlled message. The message may be any DVA/PA When Public Address announcements interrupt DVA messages this state does not change. Zone_List contains a comma separated list of Zones where DVA is active. Message_Id contains a comma separated list of dictionary item numbers playing in currently active zones.	Zone_List, Message_Id, Fault_Code
Hold	N	N/A	
Escalated	N	N/A	
Faulty	Y	A fatal fault has been detected . If the unit is still capable of being used (non-fatal fault), the fault indication will be shown in the Supplementary Field when in the idle state (see above).	Fatal_Fault_Code Fault_Code
Comms Fault	Y	Communications Exchange Server cannot communicate or detect the Device. This indicates either a network fault or failed unit.	N/A

Table 38 - Reported Network Audio Controller Device Type States

7.4.3 Operator Interface - States and Supplementary Fields

Supplementary Fields	Valid Values
Fault_Code	Refer to Table 50
Fatal_Fault_Code	Refer to Table 50
Zone_List	Comma separated list of zones where DVA/PA is active
Message_Id	Comma separated list of dictionary item numbers playing in currently active zones. This item will not be present if device is playing a PA announcement.

Table 39 - Reported Network Audio Controller Supplementary FieldsOperator Interface - States and Supplementary Fields

The API publishes the following Device States and Supplementary Fields for devices of Device Type "Operator Interface":

State	Valid	Description	Supplementary Field
Idle	Y	Device is enabled and not faulty, but not in a call or making an announcement. This state is also indicated while self health test is in progress.	Fault_Code
Isolated	Y	Device has been disabled by an external controller or UI, or location is not Active.	Fault_Code
Alerting	Y	See Table 41 below.	See Table 41 Fault_Code
Active	Y	See Table 41 below.	See Table 41 Fault_Code
Held	N	N/A	
Escalated	N	N/A	
Faulty	Y	A fault has been detected on the unit. If the unit is still capable of being used (non-fatal fault), the fault information is shown in the Supplementary Field of the Idle state.	Fatal_Fault_Code, Fault_Code
Comms Fault	Y	Communications Exchange Server cannot communicate or detect the device. This indicates either a network fault or failed unit.	N/A

Table 40 - Reported Operator Interface Device Type States

The reported state of device when alerting or active (i.e. connected to another party), is dependent on the state of calls from other devices in the system. The Table 41 below defines these situations. For example, if a call to another Operator Interface is active, and a call from a HP is alerting, the supplementary field in the call for the local Operator Interface will contain the Device ID for the HP.

Call To / From Other operator interface	Call From HP (First Received)	Call From HP(Second Onwards)	Public Address	State Reported to Control Application	Supplementary Fields
Active	Any	Any	Idle	Active	Other Operator - Device ID
Any	Active	Any	Idle	Active	None
Any	Any	Any	Active	Active	PA
Alerting	Idle	N/A	Idle	Alerting	OI_ID, Other Operator - Device ID
Idle	Alerting	Any	Idle	Alerting	HP_ID, Originating HP – Device ID
Alerting	Alerting	Any	Idle	Alerting	OI_ID, Originating Operator – Device ID, HP_ID Originating HP Device ID's

Table 41 - Alerting/Active State Determination for Operator Interface

Supplementary Fields	Valid Values
Fault_Code	Refer to Table 50
Fatal_Fault_Code	Refer to Table 50
OI_ID	DeviceId of the Operator/Crew Console the local console is communicating with. Please refer to Table 7 to see how the supplementary fields are set in various call scenarios.
HP_ID	DeviceId of the Intercom this Operator Interface is communicating with. Please refer to Table 41 to see how the supplementary fields are set in various call scenarios.
PA	Set to "Y" if this device is making a PA call. The field will not be present otherwise. Please refer to Table 41 to see how the supplementary fields are set in various call scenarios.

Table 42 - Reported Operator Interface Supplementary Fields

7.4.4 Monitor Speaker - States and Supplementary Fields

The API publishes the following Device States and Supplementary Fields for devices of Device Type "Monitor Speaker":

State	Valid	Description	Supplementary Field
Idle	Y	Device is not playing an Audio System controlled message. The supplementary field is used to display non-fatal faults with the device.	Fault_Code
Isolated	Y	Device has been disabled by an external controller or UI. When isolated, it will not play any messages.	Fault_Code
Alerting	N	N/A	
Active	Y	Device is playing an Audio System controlled message. The message may be DVA or PA when DVA/PA Monitoring is enabled on the device. The device will also be in this state when it is activated due to a call and is monitoring active calls. Zone_List is a Comma-separated list of Zones where DVA or PA is active. This list will include zero or more of zones (sink IDs) playing the message. Message_Id will be non-blank when DVA is active, and will contain a comma separated list of dictionary item numbers playing in currently active zones. PA will be set to "Y" to indicate that device is making a PA announcement. It is set to "MONITORING" if the device is monitoring the PA announcement. CallTerminals will be set when a call is being monitored, and contains the list of devices included in the call.	PA, Message_Id, Zone_List, CallTerminals, Fault_Code
Hold	N	N/A	
Escalated	N	N/A	
Faulty	Y	A fatal fault has been detected on the device. If the unit is still capable of being used (non-fatal fault), the fault indication will be shown in the Supplementary Field when in the idle state (see above).	Fatal_Fault_Code, Fault_Code
Comms Fault	Y	Communications Exchange Server cannot communicate with or detect the device on network. This indicates either a network fault or failed unit.	N/A

Table 43 - Reported Monitor Speaker Device Type States

Supplementary Fields	Valid Values
Fault_Code	Refer to Table 50
Fatal_Fault_Code	Refer to Table 50

Table 44 - Reported Monitor Speaker Supplementary Fields

7.4.5 Help Point - States and Supplementary Field

The API publishes the following Device States and Supplementary Fields for devices of Device Type “Help Point”:

State	Valid	Description	Supplementary Field
Idle	Y	Device is not playing an Audio System controlled message. The supplementary field is used to display non-fatal faults with the device.	Fault_Code
Isolated	Y	Device has been disabled by an external controller or UI. When isolated, it will not play any messages.	Fault_Code
Alerting	N	N/A	
Active	Y	Device is activated due to a call.	
Hold	Y	Device is on a call that has been put on hold.	
Escalated	N	Device is on a call that has been escalated.	
Faulty	Y	A fatal fault has been detected on the device. If the unit is still capable of being used (non-fatal fault), the fault indication will be shown in the Supplementary Field when in the idle state (see above).	Fatal_Fault_Code, Fault_Code
Comms Fault	Y	Communications Exchange Server cannot communicate with or detect the device on network. This indicates either a network fault or failed unit.	N/A

Table 45 - Reported Help Point – Device Type States

Supplementary Fields	Valid Values
Fault_Code	Refer to Table 50
Fatal_Fault_Code	Refer to Table 50

Table 46 - Reported Help Point Supplementary Fields

7.4.6 Display - States and Supplementary Field

The API publishes the following Device States and Supplementary Fields for devices of Device Type “Display”:

State	Valid	Description	Supplementary Field
Idle	Yes	Device is not Display Static Information or Video. Value will be Idle when blank screen is set.	Fault_Code
Isolated	Yes	Device has been disabled by an external controller or UI. When isolated, it will not display any messages and will be set to the blank template.	Fault_Code
Alerting	No	N/A	
Active	Yes	Device is playing video or display output.	
Hold	No	N/A	
Escalated	No	N/A	
Faulty	Yes	A fatal fault has been detected on the device. If the unit is still capable of being used (non-fatal fault), the fault indication will be shown in the Supplementary Field when in the idle state (see above).	Fatal_Fault_Code, Fault_Code
Comms Fault	Yes	Communications Exchange Server cannot communicate with or detect the device on network. This indicates either a network fault or failed unit.	N/A

Table 47 - Reported Display – Device Type States

Supplementary Fields	Valid Values
Fault_Code	Refer to Table 50
Fatal_Fault_Code	Refer to Table 50

Table 48 - Reported Display Supplementary Fields

7.4.7 Common Device Fault Codes

The Supplementary Fields Fault_Code and Fatal_Fault_Code are included in the list of supplementary fields returned by the method Device::getSupplementaryFields when there is a fault associated with the device.

These are key value pairs, where the value contains a comma separated string including mnemonics for all the faults associated with the device. The following table lists fault codes that can be included in the Fault_Code and Fatal_Fault_Code supplementary fields for all device types listed in Table 34.

Fault Mnemonic	Halts Operation	Description
BAD_CONF	No	A configuration error has been detected which could not be recovered. This fault is generated if configuration database is corrupted and there is no alternate backup configuration available.
HW_FAULT	Yes	An unrecoverable internal hardware fault has been detected. This fault is generated for detectable faults not covered by other hardware fault codes.
SELF_CHECK_ERR	No	The device specific mechanism to assess health has failed. The mechanism and fault detected by this vary by device type. Description of device specific health check implementation is covered in Section 7.5 Device Specific Self Test.
POST_ERROR	Yes	A fault is detected during power-up preventing normal device start-up

Table 49 - Common Device Fault Codes

7.4.8 Device Specific Fault Codes

The following table lists fault codes that can be included in the Fault_Code and Fatal_Fault_Code supplementary fields for specific device types. For a list of device types please refer to Table 34

Fault Code	Applicable Device Type	Halts Operation	Description
OVERHEAT	Network Audio Controller, Operator Interface, Monitor Speaker, Help Point, Display	No	The device has exceeded the maximum supported temperature
NO_VOIP_SVR	Operator Interface, Help Points	Yes	The device cannot register with the VoIP address hosted by the Communications Exchange Server
AMP_OVERHEAT_#	Network Audio Controller	No	Amplifier channel (1-N) has exceeded the maximum supported temperature. N=number of Amplifier channel
AMP_PROTECTION_#	Network Audio Controller	No	Amplifier channel (1-N) has triggered the output protection circuit as would occur if the amplifier output is short circuited or an over-voltage / over-current situation exists. N=number of Amplifier channels
AMP_FAULT_#	Network Audio Controller	No	Amplifier channel (1-N) is not oscillating or is not responding to communication requests. N=number of Amplifier channel
PSU_FAULT	Network Audio Controller	No	TGU Power Supply has failed in redundant TGU configuration.
PSU_OVERHEAT	Network Audio Controller	Yes	TGU Power Supply has exceeded the maximum supported temperature

Table 50 - Device Specific Fault Codes

7.4.9 Fault Simulation and Validation

To validate system operation, some of the faults can be initiated using the methods described in the table below.

Fault Code	Method to Generate Fault
BAD_CONF	Manually remove configuration database and backup copies if applicable. Note that user level access to this database is not provided.
HW_FAULT	User level access to generate fault is not provided
OVERHEAT	Heat the device using an external heat source Caution must be exercised to remain within storage temperature range; otherwise internal damage may occur.
NO_VOIP_SVR	Connect device to a switch/router that allows blocking IP communications (i.e. can be used as a firewall). Block communications from/to port allocation for SIP communication (default 5060) between the device and both TCX/CXS/MCX servers.
AMP_OVERHEAT_#	Heat the device using an external heat source and direct the heat towards the left side heat-sink fins. Caution must be exercised to remain within storage temperature range; otherwise internal damage may occur.
AMP_PROTECTION_#	Short circuit one (or more) of the TGU/NAC/NAM amplified output ports. Play a DVA/PA message or conduct speaker health test using the device web interface. Alternatively the SDK can also be used to initiate DVA/PA announcements.
AMP_FAULT_#	Remove components regulating self oscillation operation on amplifier or remove control bus cable to amplifier modules. Note that user level access to this database is not provided.
PSU_FAULT	Unplug one of the TGU power inputs (for TGU models with dual redundant power inputs).
PSU_OVERHEAT	Heat the device using an external heat towards the back/right corner. Caution must be exercised to remain within storage temperature range; otherwise internal damage may occur.

Table 51 - Fault Simulation and Validation

7.5 Device Specific Self Tests

The table below lists additional device specific self tests performed when Device::initiateDeviceHealthTest() is invoked.

Device Type	Device Specific Self Test
Intercoms and Help Points	Block the microphone entry to the HP with appropriate sound absorbing material and perform audio self test. Self-test is invoked automatically on system start. Self test can be manually initiated via web interface or via API using the method AudioServer::initiateDeviceHealthTest()

Table 52 - Device Specific Health Tests

7.6 Media Dictionary

The Communications Exchange Servers hold the master copy of the digital sound, image and video content. This content is collectively called the Media Dictionary. The system can be configured so that the files comprising the dictionary are automatically propagated to other devices as required in the system.

The NetSpire system includes a default media dictionary that provides the following media:

- ▶ Test tones
- ▶ Default messages
- ▶ Chime tones

During system commissioning, all other media is uploaded to the system using the NetSpire Web Administration Interface, or by using the media management facilities of the API interface.

The API interface supports adding and deleting media files in the system.

Updates to the Media Dictionary use changeset definition files, which contains a sequential list of updates to the Media Dictionary. The changeset defines a sequential list of changes that need to be applied to the existing dictionary loaded on the Communications Exchange Servers. The format and detailed contents of the changeset definition file is provided in section *7.7*.

Dictionary update can be initiated by calling the API function updateDictionary. The process to add, delete or update audio files to the system is as follows:

1. The external system copies the dictionary changeset file and media files referred in the changeset file to a location accessible by the API library, as a compressed archive file. The archive contains audio files as well as metadata including display text, description, inflection and wording for the audio file as appropriate.
2. The external system initiates an updateDictionary() API call which specifies the name of the archive containing the changeset file and media files.
3. The API library transfers the archive file to the active CXS Server. The transfer is done by the API Library and not the external system. The transfer will be achieved using FTP or SFTP (RFC 0959) as the transfer mechanism and the network will need to allow FTP/SFTP between the API Library and both CXS Servers.
4. The Communications Exchange unpacks the archive and verifies its contents.
5. The active Communications Exchange Server updates its internal dictionary by sequentially executing the actions defined in the changeset.
6. The active Communications Exchange Server distributes the changes to connected devices by applying the required changes to device dictionaries.

If only one CXS server is online (i.e. not in Comms Fault state) when the dictionary update occurs, the update will be applied to that server only. When the other Server becomes online, the server that is elected to be the active server will automatically propagate its dictionary to the secondary server.

This system can be used to create a patch to the existing dictionary as well as to force rebuild the entire dictionary from scratch by instructing the server to clear the dictionary and recreate all items. Using it as a small patch can be advantageous in situations where there is limited time or bandwidth to upload a large file to the system, e.g. using a wireless network connection where only a limited range or time is available.

When the NetSpire system dictionary is managed by an external system using the API (instead of using device administration web interfaces), it is the responsibility of the external system to ensure that both CXS Servers have the correct dictionary update level when the system starts up, and when the master/slave status of the CXS Servers switch. After system start-up, or after CXS master/slave status changes as indicated by the CXS Device State, the external system must query the dictionary revision reported by the master CXS, using the method Device:getDictionaryVersion. If the dictionary version reported by the master CXS is not what is expected by the external system, the external system needs to update the dictionary on the CXS using the API method AudioServer::updateDictionary.

7.7 Dictionary Changeset Definition File

This section specifies the file used to update the system dictionary. The file specified in this section can be passed as an argument to the AudioServer::updateDictionary() method. It is used to define a sequential list of changes that need to be applied to the existing dictionary loaded on the Communications Exchange Servers.

The dictionary changeset file must be a compressed archive file compatible with PKZIP version 2.04 and later. The filename should have a lowercase ".zip" suffix.

The compressed archive must contain the items listed below:

An XML formatted text file specifying a sequential list of changes to the dictionary. The file name, location within the archive file and content encoding must conform to the following:

- **Filename:** changeset.xml
- **Path:** / (i.e. root directory of archive file)
- **Content Encoding:** UTF-8

The XML file changeset.xml must be a well-formed and valid XML document conforming to the XML Schema Definition (XSD) provided in the Appendix (refer to Section 14.1 Dictionary Changeset XML Schema).

The XML Schema, requires that all the changes are within the element "dict-ChangeSet", and must have a namespace of "http://www.oa.com.au/asapi". One or more change definitions can be specified in the dictChangeSet in individual "change" elements.

Each change specified in the changeset must contain the following elements:

VERSIONSEQ

versionSeq: Version Sequence Number of the current change to the dictionary. It is subject to the validity checks explained below:
The versionSeq of the first change listed in the changeset must be either:

- ▶ Equal to the versionSeq of existing dictionary on the current active (primary) server+ 1. This is typically a device of the type Communications Exchange Server, as defined in Sections 7.2
- ▶ Less than the versionSeq of existing dictionary on the current active server. This defines the overlapping scenario where a changeset is provided that already contains changes applied to the existing dictionary.

In this case the active server will decide whether there is actually an overlap by examining the overlapping changes. If any conflicts are found in overlapping entries, the entire changeset will fail. If the overlapping entries are accepted as valid they will be skipped and only the changes after the overlap will be updated in the internal dictionary.

Any other value used for the versionSeq of the first change will cause the entire changeset to be ignored and will be recorded as a fail.

If a ClearDictionary command is used (see below), then the new versionSeq specified must be higher than the version number of the existing dictionary. Otherwise the entire changeset will be considered invalid and will fail.

The versionSeq of consecutive change items must be consecutive. The only exceptions to this rule are ClearDictionary and ResetDictionary operations which affect the next ex-pected versionSeq.

The dictionary version number reported by devices after executing a changeset will be updated to the versionSeq of the last executed change in the changeset. Please note Communications Exchange Server and audio output devices (NACs and MSPKRs) will have the versionSeq of "0" if there had not been any dictionary loaded yet. The current versionSeq can be reset to 0 again in the future using the ResetDictionary operation defined in the operation element of changeset.xml document. In both cases the first change defined in the changeset must have the versionSeq element set to "1".

OPERATION

operation: The operation type requested in the current change specification. It must be set to one of the following:

- ▶ UpdateDictionaryItem: Updates existing or insert new dictionary item.
- ▶ DeleteDictionaryItem: Deletes an existing dictionary item.
- ▶ ClearDictionary: Clears all items from the dictionary, and sets the current versionSeq to a new value.

- ResetDictionary: Clears all items from the dictionary and resets the new versionSeq to zero. This operation is provided as an override facility and should not be used in normal operation.

NEWVERSION

newVersion: The new version sequence number of the dictionary. The next change item specified must have a versionSeq equal to this number + 1. This parameter is required for ClearDictionary operations only.

DICTITEMNO

dictItemNo: Dictionary item number affected by the operation. This parameter is required for UpdateDictionaryItem and DeleteDictionaryItem operations only.

DICTITEM

dictItem: Updated dictionary item information, containing the XML elements listed below: . This parameter is required for UpdateDictionaryItem operation only

DEVICETYPE

deviceType: Intended device type for this dictionary item (string, valid values are: "NAC"). This specifies the device type which need to be synchronised to contain this dictionary item. Please note dictionary items with deviceType NAC are synchronised to and can be played on both NACs and MSPKRs.

CATEGORY

category: Category of the dictionary item. Dictionary items may be grouped into separate categories to facilitate for better organisation. (string, 256 char max).

AUDIOSEGMENT

audioSegment: Specifies the audio segment associated with this dictionary item. This element can be omitted for dictionary items containing displayText only. It contains the following information:

TEXT

text: The transcript of segment audio (string, 256 char max).

INFLECTION

inflection: Segments may use either a flat, falling or rising inflection depending on the position within a sentence (string, valid values are: "flat", "falling", "rising").

FILENAME

fileName: Filename of updated media file for the dictionary item. The filename must only contain alphanumeric characters, underscore and space characters. A file with this filename must be supplied in the changeset archive file. Please note the filename specified in this field is used only to provide a way of locating the audio file associated with dictionary item in the changeset archive file. The Communications Exchange Servers and audio output devices (NACs, MSPKRs) will not use the filename specified here to store audio data.

DISPLAYTEXT

displayText: The textual information to be presented on visual output devices. This element can be omitted for dictionary items containing audioSegment only (string, 512 char max).

For media files that have been referred to in the XML file, the filenames must be identical to the file-Name element specified in the changeset.xml document. The path of media files within the changeset archive file must be "media/" (i.e. media subdirectory within the compressed archive file). Media files should be using formats supported by Netspire system. Accepted formats for media files intended for use in NACs (as specified in the deviceType element in the changeset.xml document) include 16 bit mono LSB-first 48kHz PCM data with a RIFF compliant header (i.e. WAV file) and OGG Vorbis files. Accepted formats for media files intended for use in ICs must contain mono 8kHz PCMU or PCMA data with a RIFF compliant header.

An example XML document conforming to the above requirements has been provided in the Appendix (refer to Section 14.2 Sample Dictionary Changeset XML Document).

8 NetSpire SDK Class Reference – Initialisation, Status Monitoring and DVA

8.1 Overview

AudioServer class provides the primary interface to communicate with a NetSpire system.

An object of AudioServer class is intended to be created when the client application initiates a connection to the system, and the object retained during the lifetime of the connection. It can be deleted once there is no further need to communicate with the NetSpire system. It has been designed to allow creating an AudioServer object at the start of the client application, which uses this object for communications until the client application terminates.

Once an AudioServer object is created, it will attempt to connect to CXS or MCX servers specified by the customer as arguments to the connect() method. The connection will be reported as established once the AudioServer instance connects to a server with active (primary) role. Once connected to a server, the connection to all specified servers will be automatically managed by the AudioServer object. For example, if the connected server is powered off or becomes unreachable, the AudioServer will report this to the client application and will attempt to connect to another server that may have taken over the active (primary) role.

When the AudioServer instance is reporting it is connected to the system, the object can be used to perform a number of actions, including:

- ▶ Query device states
- ▶ Play either live or pre-recorded announcements, or schedule future playback
- ▶ Update media dictionary
- ▶ Retrieve references to other interfaces to manage specific aspects of the NetSpire system such as the call management, PA, or Passenger Information subsystems.

8.2 AudioServer Class Reference

The interface to the NetSpire Communications Exchange Server (CXS) is represented by the AudioServer class.

The AudioServer class should be instantiated on application start, and deleted on application completion.

8.2.1 Public Types

- typedef std::map< std::string, std::string > **KeyValueMap**
- typedef std::vector< std::string > **StringArray**
- typedef std::vector< unsigned int > **NumberArray**
- typedef std::vector< Device > **DeviceStateArray**
- typedef std::vector< DictionaryChangesetItem > **DictionaryChangesetItemArray**
- typedef std::vector< DictionaryItem > **DictionaryItemArray**
- typedef std::map< std::string, Device::DeviceClass > **DeviceClassMap**
- typedef std::vector<EventItem> **EventItemArray**
- enum **Role**
    - NONE
    - LEVEL1
    - LEVEL2

8.2.2 Member Typedef Documentation

8.2.2.1 typedef std::map<std::string, std::string> KeyValueMap

A collection of information containing string to string mappings. This is used in mapping device IDs to their IP addresses as well as the key-value pairs sent during the connect sequence.

8.2.2.2 typedef std::vector<std::string> StringArray

A collection of strings. Note: The "clear" method is required to be called when the StringArray is declared with the size specified, for example, new StringArray(8). The "clear" method needs to be called once prior to an "add" actions.

8.2.2.3 typedef std::vector<unsigned int> NumberArray

A collection of numbers. Note: The "clear" method is required to be called when the NumberArray is declared with the size specified, for example, new NumberArray(8). The "clear" method needs to be called once prior to any "add" actions.

8.2.2.4 typedef std::vector<Device> DeviceStateArray

A collection of information about zero or more devices with their current state and supplementary state information. Note: The "clear" method is required to be called when the DeviceStateArray is declared with the size specified, for example, new DeviceStateArray(8). The "clear" method needs to be called once prior to any "add" actions.

8.2.2.5 typedef std::vector<DictionaryChangesetItem> DictionaryChangesetItemArray

A collection of Dictionary Change Items (history of changes made to the Dictionary) currently stored on the Communications Exchange

8.2.2.6 typedef std::vector< DictionaryItem> DictionaryItemArray

A collection of Dictionary Items currently stored on the Communications Exchange

8.2.2.7 typedef std::map< std::string, Device::DeviceClass > DeviceClassMap

A collection of device names and their associated device classes.

8.2.2.8 typedef std::vector<EventItem> EventItemArray

A collection of EventItems that are stored in the server. Note that limit of this array can be set during AudioServer::connect by setting the KeyValuePair. In the event that a value is not set, a maximum of 1024 events will be stored after which the array will cleared of old items. Similarly when the function AudioServer::getEventList() is called, the array will be cleared at the completion of the function call.

8.2.3 Member Enumeration Documentation

8.2.3.1 enum Role

Specifies the escalation role for an Operator or Crew Console.

Enumeration

- **NONE** Location disabled
- **LEVEL1** Level 1 escalation. Indicates the Operator console is configured as the first level of escalation for IC calls.
- **LEVEL2** Level 2 escalation. Indicates the Operator console is configured as the second level of escalation for IC calls.

8.2.4 Member Function Documentation

8.2.4.1 void connect(const StringArray &serverAddresses, const KeyValueMap &config) throw (std::runtime_error, std::invalid_argument, std::out_of_range)

Connects to the active (master) Communication Exchange Server and allows a set of configuration variables to be sent as Key Value Pairs so that the system will initialise in a known state.

This method only needs to be called once at start up (and when a connection is re-established after disconnect() has been called). The API library will continuously attempt to connect to the active (master) server until a successful connection is established.

The API library will try to connect to all IP addresses in the serverAddresses list until an active (master) Communications Exchange Server is found. If only one IP address is sent to the connect method, the library will connect to the specified IP address regardless of the Master/Slave state of the specified Communications Exchange Server.

The method AudioServer::isAudioConnected() can be used to check if there is currently a connection to an Communications Exchange Server.

If the currently connected AudioServer becomes the inactive (slave) server for any reason, the API library will automatically connect to the new active (master) server.

PARAMETERS

- ***serverAddresses*** List of strings representing the IP Addresses of NetSpire CXS , MCX or TCX servers. The IP address needs to be in the IPv4 dot-decimal notation, which consists of four octets of the address expressed individually in decimal and separated by periods, for example ‘192.168.1.1’.

- ***config*** Key Value Pair List of configuration variables with format [{Config Variable, Value }]. Values incorporating commas or leading/trailing spaces must be enclosed in double quotes to delimit the argument. Any of the dynamic configuration variables listed in the Appendix (Section 12) can be specified as an argument.

  In addition to dynamic configuration variables, additional configuration items relating to the SDK can also be specified as arguments. The list of configuration item keys are shown in the following table.

Configuration Key	Description
NETSPIRE_SDK_CONFIGURE_FILE	This configuration key is deprecated and is not supported
NETSPIRE_SDK_SOCKET_PORT	Allows application to define a UDP port number to which the API will bind to. Default port number used is 20770.
NETSPIRE_SDK_CONNECT_TO_PROXY	This configuration key is for internal use only.

NETSPIRE_SERVER_HAS_CONFIG_INFO	This configuration key is for internal use only
NETSPIRE_EXPORT_LOCAL_STATE	This configuration key is for internal use only.
NETSPIRE_SDK_MAX_EVENTITEMS	Allows the application to set the maximum number of events that will be stored by the Communications Exchange Server before being discarded. The default value for this item is 1024. Refer to the function AudioServer::getEventList() for more details.

Table 53 - SDK Configuration Item Keys

EXCEPTIONS

- ▶C++ std::invalid_argument, std::out_of_range
- ▶Java, IllegalArgumentException, IndexOutOf-BoundsException
- ▶If the key value pairs are outside the bounds specified in the Appendix (Section 12), an exception will be thrown.

8.2.4.2 void disconnect()

Disconnects from the NetSpire system. This function will cause the API library to disconnect from the current server. It can be called as part of an orderly shutdown sequence or to connect to a different server. Clients can re-connect after calling disconnect() using connect().

8.2.4.3 bool isAudioConnected() const

Determines if the connection between the Communications Exchange Server API Library and a NetSpire CXS, MCX or TCX server with Active/Master role has been successfully established. In most cases the client application will not be able to retrieve information from the system, or send commands to the system unless isAudioConnected() returns “true”.

RETURNS

- ▶**bool** true if connected; false when not connected

8.2.4.4 bool isCommsEstablished() const

Determines if the communications link between the Communications Exchange Server API Library and a device it is connecting to (e.g. Communications Exchange Server, Proxy Server) has been successfully established. If the API is not currently connected to a server with Active/Master role, this method may return true while isAudioConnected() is returns false.

RETURNS

- ▶**bool** true if communications is established, false when otherwise

8.2.4.5 void setDynamicConfiguration(const KeyValueMap &dynamicConfig) throw (std::invalid_argument, std::out_of_range)

Allows a set of configuration variables to be sent as Key Value Pairs so that the system will set the system to a known state.

PARAMETERS

- ***dynamicConfig*** Key Value Pair List of configuration variables with format [{Config Variable, Value }]. Values incorporating commas, or leading/trailing spaces must use double quotations to delimit the argument. The list of accepted dynamic configuration keys and their accepted values are given in in the Appendix (Section 12).

NOTE: All supported configuration items do not need to be sent at one time using this argument. A partial or empty list can also be sent. Any configuration items not specified as part of this argument will not be modified and they will keep their current values. Their current values will be either the specified default values, or the last values assigned to them using AudioServer::connect().

8.2.4.6 KeyValueMap getDynamicConfiguration()

RETURNS

- **KeyValueMap** a list of dynamic configuration currently valid on the active Communications Exchange. If no dynamic configuration exists, an empty list will be returned.

8.2.4.7 void registerObserver(AudioServerObserver *observer)

Registers an observer with the AudioServer instance. Callback methods on the observer will be called on state changes.

PARAMETERS

- ***observer*** Observer instance

8.2.4.8 PAController * getPAController()

Get an instance of PAController which provides an interface to the PA subsystem. Please refer to Section 9.2 for more information.

RETURNS

- **PAController*** pointer to the instance of the PAController class.

8.2.4.9 CallController * getCallController()

Get an instance of Call Controller which provides an interface to the Call management subsystem. Please refer to Section 10.2 for more information.

RETURNS

- **CallController*** pointer to the instance of the CallController class.

8.2.4.10 void initiateDeviceHealthTest(const std::string &deviceId) const throw (std::invalid_argument)

Initiates health test for the specified devices.

The TEST_PENDING state will be reported by the Device::getHealthTestStatus() method immediately after AudioServer::initiateDeviceHealthTest() is called.

Once the command is received by the target device it will change its state to TEST_IN_PROGRESS and later to TEST_NONE when the test is executed or aborted for any reason.

NOTE: the state transition from TEST_IN_PROGRESS to TEST_NONE can occur very quickly in the order of milliseconds. When the state is polled, the caller should not depend on receiving TEST_IN_PROGRESS as this state may change to TEST_NONE within a very short time.

PARAMETERS

- ***deviceId*** *Mnemonic associated with a specific device in system. Please refer to Sections 7.2 for a list of device IDs and device types for a list of device IDs and device types. An empty string can be used to specify all devices, in which case health test will be initiated on all devices. Valid device type that can be specified in this argument are all device types.*

EXCEPTIONS

- ***std::invalid_argument*** (an IllegalArgumentException in Java)

8.2.4.11 void setDeviceIsolation(const std::string &deviceId, bool enable) const throw (std::invalid_argument)

Allows device to be isolated (disabled) or re-enabled. The device state will be set to ISOLATED after this method call. When isolated, the device will not be functional and most LED indications will be extinguished. It may be appropriate to isolate a device in some cases,for example all devices at a particular location may be disabled unless an authorised user has entered the area.

PARAMETERS

- . . An empty string can be used to specify all devices, in which case health test will be initiated on all devices. Valid device type that can be specified in this argument are all device types are AS, NAC, Operator Interfaces, Monitor Speaker, and IC.
- ***enable*** Isolate or re-enable device

EXCEPTIONS

- ***std::invalid_argument*** (an IllegalArgumentException in Java)

8.2.4.12 void setDeviceList(const DeviceClassMap& deviceList)

Sets the list of devices that will be returned when AudioServer::getDeviceStates() is called with an empty argument. If the returned device list is set using this method, AudioServer::getDeviceStates() will only include the devices specified in the list specified in the last invocation of AudioServer::setDeviceList and will always include these devices. If the devices are not currently connected and their attributes as included in the Device object is unknown, the attrributes will be returned as Unknown and the device state will be returned as COMMS_FAULT.

PARAMETERS

- deviceList A list of DeviceClass and Device names.

8.2.4.13 AudioServer::DeviceStateArray getDeviceStates(const std::string &deviceId = "") const throw (std::invalid_argument)

Queries states of all the devices on the NetSpire system. Also includes the supplementary information for all states.

PARAMETERS

- deviceId Mnemonic associated with a specific device in system. Please refer to Sections 7.2 for a list of device IDs and device types. An empty string can be used to specify all devices, in which case health test will be initiated on all devices. Valid device type that can be specified in this argument are AS, NAC, Operator Interfaces, Monitor Speaker, and IC.

RETURNS

- **DeviceStateArray** Information on device states for devices requested (a collection of information about zero or more devices), along with supplementary fields

EXCEPTIONS

- ***std::invalid_argument*** (an IllegalArgumentException in Java)

8.2.4.14 bool updateDictionary(const std::string &filename)

Updates the dictionary stored in the Communications Exchange Servers with the specified changeset definition file.

The Client API Library will initiate the dictionary update by transferring the changeset definition file to both Communications Exchange Servers. The transfer will be done by the API Library and not the Client. The transfer will be achieved using FTP as the transfer mechanism and the network. The command will fail and return false if both Communications Exchange Servers are not available at the time.

The Communications Exchange Servers will then unpack the changeset file and verify its contents. Please see Section 1.8 Dictionary Changeset Definition File for the naming and contents requirements of the file. The changeset defines a sequential list of changes that need to be applied to the existing dictionary loaded on the Communications Exchange Servers. It can be used to create a patch to the existing dictionary as well as to force rebuild the entire dictionary from scratch by instructing the server to clear the dictionary and recreate all items. Using it as a small patch can be advantageous in situations where there is limited time or bandwidth to upload a large file to the system e.g. using a wireless network connection where only a limited range or time is available.

The Communications Exchange Servers internally manage a similar list of dictionary changes. Once the Communications Exchange Servers verify the validity of the new changeset, they will update their internal dictionary by sequentially executing the actions defined in the changeset. They will then distribute the changes to connected devices by applying the required changes to device dictionaries. Once the process is successfully completed, the Communications Exchange Servers and all devices in the system will have the same dictionary items and media files, and they will report identical dictionary versions.

The dictionary version number reported by devices after executing a changeset will be updated to the versionSeq of the last executed change in the changeset. Please note Communications Exchange Server and audio output devices (NACs and MSPKRs) will have the versionSeq of "0" if there had not been any dictionary loaded yet. The current versionSeq can be reset to 0 again in the future using the ResetDictionary operation defined in the operation element of changeset.xml document.

If AudioServer.updateDictionary fails for any reason the dictionary update will be abandoned and the existing dictionary and reported dictionary version number will not be affected

Dictionary item number range 99000 to 99999 is reserved for the default dictionary to store test tones, built-in messages and chimes. Note the system retains the default dictionary; and if an individual item is overwritten it will simply play the file provided instead of the built-in default. This ensures if no default files are overwritten in a subsequent update the system will revert back to using the default dictionary.

PARAMETERS

- ▶ ***filename*** Filename (including full system path) of changeset definition file. The contents of the file must conform to the specification given in Section 1.8 Dictionary Changeset Definition File.

RETURNS

- ▶ **bool** false and aborts operation if both Communications Exchange Servers are not online at the time the command is received. Returns true if both Communications Exchange Servers are online.

8.2.4.15 AudioServer::DictionaryChangesetItemArray getDictionaryChangeset()

Returns the current version history of the media dictionary.

RETURNS

- ▶ **DictionaryChangesetItemArray** list of Dictionary Changeset Item containing the full list of version history

8.2.4.16 AudioServer::DictionaryItemArray getDictionaryItems()

Returns the content of the current media dictionary that is stored in the Communications Exchange Server.

RETURNS

- ▶ **DictionaryItemArray** list of Dictionary Items

8.2.4.17 void setOutputGain(const Gain &gain) throw (std::invalid_argument, std::out_of_range)

Set default output gain on network amplifiers. Gain adjustment will be applied to all amplifiers equally including inductive loops. This will affect DVA and PA gain for all announcements Zones.

PARAMETERS

- ▶ ***dB*** Output gain in decibels.

EXCEPTIONS

- ▶ ***std::invalid_argument, std::out_of_range*** (In Java, IllegalArgumentException or IndexOutOf-BoundsException respectively)

8.2.4.18 Gain getOutputGain(const std::string &deviceId) const throw (std::invalid_argument)

Query default output gain on all network amplifiers including inductive loop amplifiers. This gain setting affects DVA and PA gain for all announcements Zones.

PARAMETERS

- deviceId Mnemonic associated with a specific device in system. Please refer to Sections 7.2 for a list of device IDs and device types. An empty string can be used to specify all devices, in which case health test will be initiated on all devices. Valid device type that can be specified in this argument is NAC only.

EXCEPTIONS

- ***std::invalid_argument*** (In Java, IllegalArgumentException)

RETURNS

- **Gain** Output gain in decibels

8.2.4.19 AudioServer::Role getRole(const std::string &deviceId) const throw (std::invalid_argument)

- Returns the current escalation level of the device. Note the same role will be assigned to all Operator Interface in one cabin.

PARAMETERS

- deviceId Mnemonic associated with a specific device in system. Please refer to Sections 7.2 for a list of device IDs and device types. Valid device type that can be specified in this argument is NAC and CI only.

RETURNS

- Role of the device (LEVEL1, LEVEL2 or NONE)

EXCEPTIONS

std::invalid_argument (In Java, IllegalArgumentException)

8.2.4.20 void uploadFile(const std::string &filename)

Allows a file to be uploaded to the connected Communications Exchange Server. Note that in a fail over system, the file will only be uploaded to the active Communications Exchange Server (or to the Communications Exchange Server that the API is connected to).

This functionality is used by some API client applications that make use of loadable modules installed on Communications Exchange Servers.

PARAMETERS

- ***filename*** Filename (including full system path) of the fle that will be uploaded to the Communications Exchange Server.

RETURNS

- This function does not return anything.

8.2.4.21 AudioServer::EventItemArray getEventList()

Returns a list of all unread events raised and stored in the connected Communications Exchange Server. This allows a client application can poll the stored event list to get a list of changes occurred since last poll.

The Communications Exchange Server will maintain a maximum of 1024 events after which the events will be discarded and internal cache is reinitialized. It is possible to change the maximum number of events stored by the Communications Exchange Server at time of connection using the connection parameter NETSPIRE_SDK_MAX_EVENTITEMS.

Note that the internal cache will also be cleared at the completion of this function call.

RETURNS

- **AudioServer::EventItemArray** returns the current list of events. Note that the internal cache is cleared at the completion of this function call.

8.2.4.22 std::string getSystemRevision()

Returns the version number for the entire system. System version is commonly expressed as a string value with the format: XX.YY.ZZ where XX, YY and ZZ are integer values.

Note this method will return a blank string if called immediately after calling AudioServer::connect(). A non-empty system revision will be returned once the API library retrieves information on the system revision.

RETURNS

- **std::string** indicating the system version.

8.2.4.23 bool isSystemRevisionConsistent()

Indicates if all the devices within the system have the correct revision.

This function can temporarily return false while software upgrade is in progress and should not occur during normal operation as devices are automatically updated to the current revision by the CXS/TCX/MCX when a component is replaced. If the function returns false during normal operation, it indicates the automatic configuration management system is not operating correctly.

RETURNS

- bool to indicate the status of the system revision. Value of ‘true’ is returned if all devices revision controlled by the Communications Exchange have the correct software revision. Value of ‘false’ is returned if a subset of the devices revision controlled by the Communications Exchange has a different software vision.

8.2.4.24 std::string getSDKRevision()

Returns the current version of the Netspire SDK. The format of the string is XX.YY.ZZ where XX, YY and ZZ are integer values

RETURNS

- **std::string** indicating the Netspire SDK version.

8.2.4.25 void initiateShutdown() const

Provides an indication to Communication Exchange that power may be removed without other notification. This function does not shutdown or restart the device. The facility to restart the device is provided by the initiateRestart() method..

8.3 AudioServerObserver Class Reference

AudioServerObserver class provides an interface for client applications to be notified of events that occur in the NetSpire system.

The client application needs to create a child class inheriting from AudioServerObserver, and override the methods the application is interested in.

Note the methods of the observer class will be executed in the context of a different thread, and the client applications needs to ensure all actions taken within the observer methods are thread-safe. In addition, all observer methods are executed within the same thread, observer methods are not intended to block. Recommended usage of the observer methods is to send a message to another thread in the client application to perform any long tasks.

8.3.1 Detailed Description

The AudioServerObserver provides an interface to observe changes in the audio system including device status changes.

8.3.2 Constructor and Destructor

8.3.2.1 virtual AudioServerObserver::˜AudioServerObserver()

Callback function - called when deleting the observer object

8.3.3 Member Function Documentation

8.3.3.1 virtual void AudioServerObserver::onCommsLinkUp()

Callback function - called when connection to a device is established. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***device*** Reference to the device that has changed state

8.3.3.2 virtual void AudioServerObserver::onCommsLinkDown()

Callback function - called when connection to a device is lost. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***device*** Reference to the device that has changed state

8.3.3.3 virtual void AudioServerObserver::onAudioConnected()

Callback function - called when connection to an active server is established. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

▶ ***device*** Reference to the device that has changed state

8.3.3.4 virtual void AudioServerObserver::onAudioDisconnected()

Callback function - called when connection to an active server is lost. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

▶ ***device*** Reference to the device that has changed state

8.3.3.5 virtual void AudioServerObserver::onDeviceStateChange(Device∗ device)

Callback function - called when a Device state is updated. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

▶ ***device*** Reference to the device that has changed state

8.3.3.6 virtual void AudioServerObserver::onDeviceDelete(Device∗ device)

Callback function - called when a Device is deleted from the system. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

▶ ***device*** Reference to the device that is deleted from the NetSpire AudioServer.

8.3.3.7 virtual void AudioServerObserver::onVUMeterUpdate(int level)

Callback function - called when VU level is updated during PA. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

▶ **level** Level of input as received from the DSP

8.3.3.8 virtual void AudioServerObserver::onConfigUpdate(std::string progress, std::string key, std::string value)

Callback function indicating a configuration value is updated. This method will be executed in a separate thread created by the SDK library.

8.3.3.9 virtual void AudioServerObserver::onStateUpdate(bool isImportData, int key, std::string dataMsg)

Callback function indicating the state of connected device value is updated - This method will be executed in a separate thread created by the SDK library.

8.3.3.10 virtual void AudioServerObserver::onRealTimeItemUpdate(std::string key, std::string value)

Callback function indicating a real-time configuration value is updated - This method will be executed in a separate thread created by the SDK library.

8.3.3.11 virtual void AudioServerObserver::onDebugMessage(std::string msg)

Callback function indicating the SDK library has published a debug message. The value of the message can be useful while developing client applications. This method will be executed in a separate thread created by the SDK library.

8.4 Device Class Reference

Device class provides both static and state information on devices within the NetSpire system.

A list of devices within the NetSpire system can be queried using the method AudioServer::getDeviceStates().

8.4.1 Public Types

- typedef std::map< std::string, std::string > **SupplementaryFields**
- typedef std::vector< LogEntry > LogEntryList
- enum **State**
  - IDLE = 0,
  - ALERTING,
  - ACTIVE,
  - HELD,
  - ESCALATED,
  - ISOLATED,
  - FAULTY,
  - COMMSFAULT,
  - INACTIVE
- enum **HealthTestStatus**
  - TEST_PENDING = 0,
  - TEST_NONE,
  - TEST_IN_PROGRESS
- enum **DictionaryUpdateStatus**
  - DICT_UPDATE_NONE = 0,
  - DICT_UPDATE_IN_PROGRESS =1 ,
  - DICT_UPDATE_FAILED =2 ,
  - DICT_UPDATE_PENDING =3
- enum **DeviceClass**
  - UNKNOWN_DEVICE = 0,
  - COMMUNICATIONS_EXCHANGE,
  - NETWORK_AUDIO_CONTROLLER,
  - OPERATOR_CONSOLE,
  - MONITOR_SPEAKER,
  - HELP_POINT,
  - PASSENGER_INFORMATION_DISPLAY,
  - TRAIN_RADIO

8.4.2 Detailed Description

Device class provides both static and state information on devices within the NetSpire system.

A list of devices within the NetSpire system can be queried using the method AudioServer::getDeviceStates().

The Device class includes static information includes model name, device capabilities, device location (as configured in the administration web UI). Dynamic information includes device activity state, supplementary fields, health test status, dictionary status and version and firmware revision.

8.4.3 Member Typedef Documentation

8.4.3.1 typedef std::map<std::string, std::string> Device::SupplementaryFields

A collection of key value pairs for storing device state supplementary fields.

Please refer to Section 7.4 Device States and Supplementary Fields for the list of supported supplementary fields for each device type.

8.4.4 Member Enumeration Documentation

8.4.4.1 enum Device::State

Defines a device state. Note that not all states are necessarily possible for each device type.

ENUMERATION

- ▶**IDLE** No call in progress, device is available
- ▶**ALERTING** Processing a call request, no call exists yet
- ▶**ACTIVE** A call is in progress
- ▶**HELD** The call at this device is on hold
- ▶**ESCALATED** A Call has an escalated call
- ▶**ISOLATED** The device is unable to function or respond
- ▶**FAULTY** The device is fully or partly reporting a faulty status. The other Device
- ▶**COMMSFAULT** Device unreachable, state can't be determined
- ▶**INACTIVE** Device is functioning in a limited mode

8.4.4.2 enum Device::HealthTestStatus

Specifies the state of the most recent health test request

ENUMERATION

- **TEST_PENDING** AudioServer::initiateDeviceHealthTest() has been called to initiate a test. The target device has not yet responded to the command.
- **TEST_NONE** No health tests initiated or the last health test is completed
- **TEST_IN_PROGRESS** Last health test is in progress (queued or being executed)

8.4.4.3 enum Device::DictionaryUpdateStatus

Specifies the state of the most recent dictionary update request

ENUMERATION

- **DICT_UPDATE_NONE** No dictionary updates initiated or the last update is completed. Please query the dictionary version using getDictionaryVersion() to verify the expected dictionary version is loaded on the device. Currently reported dictionary version can continue to be used in announcement requests.
- **DICT_UPDATE_IN_PROGRESS** Last dictionary update is in progress. Media files are being replaced in this state and behaviour of announcement requests while the device is in this state is undefined.
- **DICT_UPDATE_FAILED** Last dictionary update failed. Currently reported dictionary version can continue to be used in announcement requests.
- **DICT_UPDATE_PENDING** Dictionary update is pending and is being processed in the background. Currently reported dictionary version can continue to be used in announcement requests.

8.4.4.4 enum Device::DeviceClass

Specifies the class that the device belongs to

ENUMERATION

- **UNKNOWN_DEVICE** Devices that do not fit any of the category.
- **COMMUNICATIONS_EXCHANGE** Devices that are responsible centralized functions and controls. These include CXS, TCX and MCX
- **NETWORK_AUDIO_CONTROLLER** Devices that act as network attached audio. These include TGU, NAC, NAM, NAR.

- **OPERATOR_CONSOLE** Devices that allow operators to interact with the end user. These include IPPA, CI, CP, CC
- **MONITOR_SPEAKER** Devices that are used for monitoring of audio within the system. These include CAC, MSPK
- **HELP_POINT** Devices that are used for end user interaction. These include IC, PEI, PECU, HP
- **PASSENGER_INFORMATION_DISPLAY** Devices that are used to display information to the end user. These include IDI, EDI, EIDI, VCU
- **TRAIN_RADIO** Rolling stock wireless communications devices. Acronym used for this device class is TR.

8.4.5 Member Function Documentation

8.4.5.1 std::string Device::getName() const [virtual]

Returns the name of the device.

RETURNS

- the device name

8.4.5.2 Device::State Device::getState() const [virtual]

Returns the current state of this device.

RETURNS

- the device State

8.4.5.3 std::string Device::getStateText() const

Returns the current state of this device in a human-readable form.

RETURNS

- text describing the current device State.

8.4.5.4 Device::SupplementaryFields Device::getSupplementaryFields() const

Returns the supplementary fields of this device state.

RETURNS

- SupplementaryFields provided for the device.

8.4.5.5 Device::HealthTestStatus Device::getHealthTestStatus() const

Query the status of the most recent health test requested.

8.4.5.6 Device::initiateDeviceHealthTest().

The TEST_PENDING state will be reported by the Device::getHealthTestStatus() method immediately after Device::initiateDeviceHealthTest() is called.

Once the command is received by the target device it will change its state to TEST_IN_PROGRESS and later to TEST_NONE when the test is executed or aborted for any reason.

NOTE: the state transition from TEST_IN_PROGRESS to TEST_NONE can occur very quickly in the order of milliseconds. When the state is polled, the caller should not depend on receiving TEST_IN_PROGRESS as this state may change to TEST_NONE within a very short time.

RETURNS

- ▶A collection of health test status for devices queried.

8.4.5.7 bool Device::getDictionarySupport() const

Query the device to see if dictionary is supported on the device.

RETURNS

- ▶True/false to indicate if dictionary is supported.

8.4.5.8 unsigned int Device::getDictionaryVersion() const

Query the version number of the dictionary maintained by the CXS, NACs, and MSPKs. This method will always return 0 for other devices. To query the dictionary version of the entire audio system, the CSS can call this method on the primary Communications Exchange Server. The primary Communications Exchange Server will ensure the rest of the devices are updated to the dictionary version loaded on itself.

RETURNS

- ▶Version number of the dictionary stored on the device

8.4.5.9 Device::DictionaryUpdateStatus Device::getDictionaryUpdateStatus() const

Query the status of the most recent dictionary update requested using AudioServer::updateDictionary(). This method will always return DICT_UPDATE_NONE if the device does not support the media dictionary.

RETURNS

- ▶The status of the most recent dictionary update.

8.4.5.10 bool Device::getInputState(int inputNo) const

Retrieves the externally set input state on the given device. Valid device types that can be queried with this method are CXS, NAC and IC.

PARAMETERS

- ▶***inputNo*** Numeric integer specifying one of the inputs on the device.

RETURNS

- ▶True if the input state is set on the device. False for all other cases, including when the specified input number is not supported on the device.

8.4.5.11 bool Device::getOutputState(int outputNo) const

Retrieves the internally set output state on the given device. Valid device types that can be queried with this method are CXS, NAC and IC.

PARAMETERS

- ▶***inputNo*** Numeric integer specifying one of the inputs on the device.

RETURNS

- ▶**bool** True if the output state is set on the device. False for all other cases, including when the specified output number is not supported on the device.

8.4.5.12 void Device::setOutputState(unsigned int outputNo, unsigned int state)

Sets the output state on the given device.

PARAMETERS

- ▶***outputNo*** Numeric integer specifying one of the outputs on the device.
- ▶***state*** 1 signifies asserting the output, 2 negating the output

8.4.5.13 int Device::getDstNo()

Retrieves the destination number that the device is currently set to. The destination number is generated using the location Id and the device Id.

RETURNS

- ▶**int** Destination number of the device

8.4.5.14 int Device::getPortNo()

Retrieves the port number that the device is set to.

RETURNS

- ▶ Port number of the device

8.4.5.15 std::string Device::getIP() const

Retrieves the IP address of the device.

RETURNS

- ▶ IP address of the device.

8.4.5.16 std::string Device::getSoftwareRevision() const

Retrieves the version number of the installed firmware. The firmware version has the format XX.YY.ZZ where XX indicates major number, YY indicates minor number and ZZ indicates patch number.

RETURNS

- ▶ Firmware revision that is installed on the device.

8.4.5.17 std::string Device::getLocationName() const

Retrieves the location name that is set when the device is configured.

RETURNS

- ▶ Name of device location

8.4.5.18 std::string Device::getLocationId() const

Retrieves the location ID that is set when the device is configured. Format of location Id is “xxx”. Location Id is entered by the operator when the device is configured.

RETURNS

- ▶ Location Id of the device.

8.4.5.19 std::string Device::getDeviceIndex() const

Retrieves the device index that is set when the device is configured. Format of device Id is “yyy”. Device Index is is entered by the operator when the device is configured.

RETURNS

- ▶ Device index of the device.

8.4.5.20 DeviceClass Device::getDeviceClass() const

Retrieves the class that the device belongs to. Refer to Section 8.4.4.4 which lists supported device classes.

RETURNS

- ▶ Device class of the device.

8.4.5.21 DeviceModel Device::getDeviceModel() const

Retrieves the model representing the device. Refer to Section **Error! Reference source not f ound.** for detailed information.

RETURNS

- ▶ Model representing the device.

8.4.5.22 LogEntryList Device::getDeviceLog() throw std::invalid argument

Retrieves the device log,which is comprised ofa list of log entries. Note device log of any device other than the connected Active CXS is not made available by the API, and an exception will be thrown.

RETURNS

- ▶ List of LogEntry objects in the device log.

EXCEPTIONS

std::invalid_argument (In Java, IllegalArgumentException) The Device Log of a devive other than the connected Active CXS is requested.

8.4.5.23 void Device::appendToDeviceLog(const LogEntry& entry)

Appends a new log entry to the device log. Note the device log may be configured to have a finite list of elements, and inserting a new entry may result in the earliest log item to be removed. Also note device log of any device other than the connected Active CXS is not made available by the API, and an exception will be thrown.

The timestamp and IP Address fields of the log entry will be automatically assigned by the system, and the content of these fields in the entry argument will not be included in the device log.

PARAMETERS

- ▶ ***entry*** Log entry to be added to the device log.

EXCEPTIONS

std::invalid_argument (In Java, IllegalArgumentException) The Device Log of a device other than the connected Active CXS is requested to be appended.

8.4.5.24 void Device::initiateRestart() const

Undertakes a managed restart of the device. .Note that the device restart may disrupt active operations resulting in user visible service disruption. The resart timing is determined by each device and may not occur immediately, but at the earlier possible occasion where the device is able to safely proceed.

8.4.5.25 void Device::setCondition(const std::string& conditionName, bool value)

Sets the value of a condition variable. NetSpire devices can be configured to react to and take a specified list of actions based on user settable or event based conditions.

PARAMETERS

- ▶ ***conditionName*** Name of the condition variable. If the condition is not present it will be created and then assigned a value.
- ▶ ***value*** Value to set the condition variable.

8.4.5.26 void Device::setCondition(const std::string& conditionName, const std::string& value)

Sets the value of a condition variable. NetSpire devices can be configured to react to and take a specified list of actions based on user settable or event based conditions.

PARAMETERS

- ▶ ***conditionName*** Name of the condition variable. If the condition is not present it will be created and then assigned a value.

8.4.5.27 *value* Value to set the condition variable.bool Device::getCondition(const std::string& conditionName) const

Returns the value of a condition variable. NetSpire devices can be configured to react to and take a specified list of actions based on user settable or event based conditions.

PARAMETERS

- ***conditionName*** Name of the condition variable.

RETURNS

- Value of the condition variable. If the condition is not present, “false” will be returned.

8.5 DeviceModel Class Reference

DeviceModel class can be used to get information on some attributes of a Device. Among the information that can be retrieved are device model name, number of DIOs and number of audio output channels present on devices of this model.

8.5.1 Detailed Description

DeviceModel class provides additional information about the device. Information includes model name, number of digital inputs, number of digital outputs and number of audio output channels.

Digital input count, output count and analog audio input and output counts can be retrieved by accessing this class.

8.5.2 Constructor and Destructor Documentation

8.5.2.1 DeviceModel::DeviceModel()

Constructor

8.5.2.2 ~DeviceModel::DeviceModel()

Destructor

8.5.3 Member Function Documentation

8.5.3.1 std::string DeviceModel::getName() const

Returns the model of the device.

RETURNS

▶ ***string*** value indicating the model of the device. An empty string will be returned if the device is not configured properly or the system is not able to determine the model of the device.

8.5.3.2 int DeviceModel::getDigitalInputCount() const

Returns the total number of digital inputs that is supported by the device.

RETURNS

▶ ***integer*** value representing the total number of digital inputs.

8.5.3.3 int DeviceModel::getDigitalOutputCount() const

Returns the total number of digital outputs that is supported by the device.

RETURNS

- ***integer*** value representing the total number of digital outputs.

8.5.3.4 int DeviceModel::getAudioOutputChannels() const

Returns the total number of audio output channels that is supported by the device.

RETURNS

- **integer** value representing the total number of audio output channels.

8.6 DictionaryChangesetItem Class Reference

DictionaryChangeItem is used to store a single change to the Media Dictionary. Client applications can call the function AudioServer::getDictionaryChangeset() to get the last dictionary update that was performed on the Communications Exchange Server, which includes a list of DictionaryChangeItem objects.

8.6.1 Public Attributes

- int **key**
  Indicates a unique number used by the Communications Exchange Server. This is used internally and has no direct relevance to the client application
- int **versionSeq**
  Indicates the version sequence number for the dictionary change item
- std::string **operationId**
  Indicates the operation id of the dictionary change item. Section 1.8 indicates the supported operations
- int **newVersion**
  Indicates the new version number that the dictionary will be reset to. Note that this is only applicable ClearDictionaryoperation
- int **itemNo**
  Indicates the unique dictionary item number
- std::string **deviceType**
  Indicates the device type for the dictionary item. This field is obsolete.
- std::string **description**
  A short description of the dictionary item
- std::string **category**
  Indicates the category for the dictionary item
- std::string **audioSegment**
- std::string **image**
- std::string **video**
- std::string **displayText**
  Text to be displayed on the passenger information displays
- std::string **metadata**
  Storage for additional data for the dictionary (internal use only)
- int **lastUpdate**
  Storage for additional data for the dictionary (internal use only)

8.6.2 Detailed Description

DictionaryChangeItem is used to store a single change to the Media Dictionary. Client applications can call the function AudioServer::getDictionaryChangeset() to get the last dictionary update that was performed on the Communications Exchange Server, which includes a list of DictionaryChangeItem objects.

The structure included in the DictionaryChangeItem is identical to the structure used in the XML formatted document that is used to update dictionary using AudioServer::updateDictionary (refer to Section 8.2.4.14). For more information on dictionary, dictionary structure and how it is managed, please refer to Sections 7.6 Media Dictionary and 7.7 Dictionary Changeset Definition File.

8.6.3 Member Function Documentation

DictionaryChangeItem has no member functions.

8.7 DictionaryItem Class Reference

DictionaryItem is used to store the contents of a dictionary. It contains information about the dictionary item no, the type of dictionary item (audio, image or video) and the files that are associated with this dictionary item.

The list of dictionary items available on the system can be queried using AudioServer::getDictionaryItems(). The dictionary items can then be either played immediately usign PAController::playMessage(), or scheduled for playback in the future using PAController::createSchedule().

8.7.1 Public Attributes

- int **itemNo**
  Indicates the unique dictionary item number
- std::string **description**
  A short description of the dictionary item
- std::string **category**
  Indicates the category for the dictionary item
- std::string **displayText**
  Text to be displayed on the passenger information displays
- std::string **fileList**

  A string value indicating the filename, type, duration (ms) and size (bytes) for the audio, image or video segments. The string can include multiple files formatted according to the following rule: filename^filetype^durationMS^fileSizeBytes!....!filename^filetype^durationMS^fileSizeBytes

  If dictionary item is of type audio then filetype will be audio_flat or audio_falling or audio_rising.

  If dictionary item is of type image, then filetype will be bitmap.

  If dictionary item is of type video, then filetype will be video_generic.
- std::string **metadata**
  Storage for additional data for the dictionary (internal use only)

8.7.2 Detailed Description

DictionaryItem is used to store the contents of a dictionary. It contains information about the dictionary item no, the type of dictionary item (audio, image or video) and the files that are associated with this dictionary item. Each DictionaryItem will have a unique id and this id will be used by the Communications Exchange Server to play the dictionary content within the system.

8.7.3 Member Function Documentation

8.7.3.1 std::vector<std::string> DictionaryItem::getContentsURI() const

Requests access to the media contents stored in the specified Dictionary Item.

Each DictionaryItem can contain zero or more media files. The public attribute **fileList** contains the list of media files stored for the DicitonaryItem.

This method returns a list of HTTP URIs for each media file stored for the DictionaryItem. The list can have zero or more entries.

RETURNS

- ***vector<string>*** a list of HTTP URIs for each media file stored for the DictionaryItem. The list can have zero or more entries.

8.8 EventItem Class Reference

EventItem is used to store information when an event is raised. The function AudioServer::getEventList() is used to get all the events that are stored in the server. EventItem will indicate the date and time when the event was raised along with the type of event and additional information depending on the type of event raised.

8.8.1 Public Attributes

- enum **EVENT_TYPE**
  - COMMS_LINK_UP=0,
  - COMMS_LINK_DOWN,
  - AUDIO_SERVER_CONNECTED,
  - AUDIO_SERVER_DISCONNECTED,
  - DEVICE_STATE_CHANGED,
  - DEVICE_DELETED,
  - PA_SOURCE_UPDATED,
  - PA_SOURCE_DELETED,
  - PA_SINK_UPDATED,
  - PA_SINK_DELETED,
  - PA_TRIGGER_UPDATED,
  - PA_TRIGGER_DELETED,
  - PA_SELECTOR_UPDATED,
  - PA_SELECTOR_DELETED,
  - CALL_UPDATED,
  - CALL_DELETED,
  - CDR_MESSAGE_UPDATED,
  - CDR_MESSAGE_DELETED,
  - TERMINAL_UPDATED,
  - TERMINAL_DELETED
- std::string **dateTime**
  Indicates the date and time when the event was raised. Format of the string is year-month-day hour:min:sec
- EVENT_TYPE **type**
  Indicates the type of event that was raised. Events can be one of the type indicated in the enum EVENT_TYPE
- int **id**
  Indicates the ID of the raised event. Note that the id is set to:
  - *0 for COMMS_LINK_UP, COMMS_LINK_DOWN, AUDIO_SERVER_CONNECTED and AUDIO_SERVER_DISCONNECTED*
  - device id for DEVICE_STATE_CHANGED and DEVICE_DELETED

  - souce id for PA_SOURCE_UPDATED and PA_SOURCE_DELETED
  - sink id for PA_SINK_UPDATED and PA_SINK_DELETED
  - trigger id for PA_TRIGGER_UPDATED and PA_TRIGGER_DELETED
  - selector id for PA_SELECTOR_UPDATED and PA_SELECTOR_DELETED
  - call id for CALL_UPDATED and CALL_DELETED
  - cdr id for CDR_MESSAGE_UPDATED and CDR_MESSAGE_DELETED
  - term id for TERMINAL_UPDATED and TERMINAL_DELTED

- Device device
  Includes information about the device when the event is of type DEVICE_STATE_CHANGED and DEVICE_DELETED
- PaSource **paSource**
  Includes information about PaSource when the event is of type PA_SOURCE_UPDATED or PA_SOURCE_DELETED
- PaSink **paSink**
  Includes information about PaSink when the event is of type PA_SINK_UPDATED or PA_SINK_DELETED
- PaTrigger **paTrigger**
  Includes information about PaTrigger when the event is of type PA_TRIGGER_UPDATED or PA_TRIGGER_DELETED
- PaSelector **paSelector**
  Includes information about PaSelector when the event is of type PA_SELECTOR_UPDATED or PA_SELECTOR_DELETED
- CallInfo **callInfo**
  Includes information about CallInfo when the event is of type CALL_UPDATED or CALL_DELETED
- CDRInfo **cdrInfo**
  Includes information about CDRInfo when the event is of type CDR_MESSAGE_UPDATED or CDR_MESSAGE_DELETED
- TerminalInfo **termInfo**
  Includes information about the TerminalInfo when the event is of type TERMINAL_UPDATED or TERMINAL_DELETED

8.8.2 Detailed Description

EventItem is used to store information when an event is raised. The function AudioServer::getEventList() is used to get all the events that are stored in the server. EventItem will indicate the date and time when the event was raised along with the type of event and additional information depending on the type of event raised.

If an event is raised which is of type COMMS_LINK_UP, COMMS_LINK_DOWN, AUDIO_SERVER_CONNECTED or AUDIO_SERVER_DISCONNECTED have no additional information associated with them. Under such condition, it is adviced not use use data memebers device, paSource, paSink, paTrigger, paSelector, callInfo, cdrInfo and termInfo as they will contain invalid data.

8.8.3 Member Function Documentation

EventItem has no member functions.

8.9 Gain Class Reference

Gain Class is used as a parameter when setting gain levels on audio sinks (via PaSink::setGain) and zones and retrieving current gain levels (via PaSink::getGain).

8.9.1 Detailed Description

Specifies the amplifier gain level used in audio sinks and zones. The "level" field indicates a numeric value in dBs.

8.9.2 Constructor and Destructor Documentation

8.9.2.1 Gain::Gain (double level)

Initializes the Gain object to the specified level.

PARAMETERS

- ***level*** Gain level. System limits levels to the range [-96, 0] dB. Values exceeding this range will be converted to the closest value within the range.

8.9.3 Member Function Documentation

8.9.3.1 void Gain::setLevel (double level)

Sets gain level.

PARAMETERS

- ***level*** Gain level. System limits levels to the range [-96, 0] dB. Values exceeding this range will be converted to the closest value within the range.

8.9.3.2 double Gain::getLevel() const

Returns gain level.

RETURNS

- Gain level.

8.10 LogEntry Class Reference

LogEntry is used to represent individual log entries in the device log. The Device Log is maintained by each device and is a collection of individual log entries.

The list of LogEntries in the Device Log can be queried using Device::getDeviceLog(). A new log entry can be appended to the Device Log using Device::appendToDeviceLog().

8.10.1 Public Attributes

- int **itemNo**
  Indicates the unique item number for the log entry
- std::string **time**
  Date/time associated with the log entry in text format
- std::string **address**
  Originating address of the connection that has created the log entry. This is typically an IPv4 or IPv4 address.Most log entries are created by connections within the device itself and the address will be the address of the device at the time the log entry is created.
- std::string **deviceName**
  Contains the name of device at the time the log entry is created.
- std::string **location**
  Contains the location of the device at the time the log entry is created.
- std::string **eventCategory**
  Indicates the event category of the log item
- std::string **eventID**
  Indicates the event ID of the log item
- std::string **eventParameters**
  Contains the event parameters and description associated with the logged event

8.10.2 Detailed Description

LogEntry is used to store the contents of each entry in the Device Log. It contains information on the type of event, details on event including date/time it occurred. The information is returned as strings.

8.10.3 Member Function Documentation

LogEntry has no member functions.

8.11 Message Class Reference

Message Class represents message to play on audio and visual devices, and is used as a parameter when playing a message via PAController::playMessage.

8.11.1 Public Types

- enum **Type**
  - LIVE_STREAM = 0,
  - RECORDED_STREAM =1 ,
  - DICTIONARY = 2,
  - DISPLAY = 3,
  - TTS = 4,
  - BGM = 5,
  - NONE = 7,
  - TEMPLATE = 8
- enum **TextEncoding**
  - ISO_8859_1 = 0,
  - UTF_8
- enum **AudioEncoding**
  - L16_48K = 0,
  - L16_16K,
  - L24_48K,
  - L24_16K

8.11.2 Detailed Description

Message Class represents message to play on audio and visual devices, and is used as a parameter when playing a message via PAController::playMessage. Allows flexible creation of complicated messages that can be a combination of dictionary items, TTS created audio, and audio or visual messages created in accordance to a predefined template.

8.11.3 Member Enumeration Documentation

8.11.3.1 enum AnnouncementType

Specifies the type for an audio or visual message.

Enumeration

- ▶**LIVE_STREAM** PA audio streamed using an IP connection, where the audio source is live, including analogue or network microphones

- ▶**RECORDED_STREAM** PA audio streamed using an IP connection, where the audio source is recorded, including user recorded messages, previewed live PA, previewed TTS, streamed DVA
- ▶**DICTIONARY** Message is played from contents of local media dictionary
- ▶**DISPLAY** Display control or update message
- ▶**TTS** PA audio streamed using an IP connection, where the audio content is generated using a TTS (Text to Speech) engine
- ▶**BGM** Background music audio streamed using an IP connection
- ▶**NONE** No audio or display content included in the message
- ▶**TEMPLATE** Message is created in accordance to a pre-defined template. Templates provide a structure for the message, and specify a number of variables that are substituted at playback time. The values for the template variables are sent to the SDK using a list of string key value pairs.

8.11.3.2 enum TextEncoding

Specifies the encoding of specified text.

Enumeration

- ▶**ISO_8859_1** 8 bit latin encoding for Western European languages as specified by ISO 8859-1, also known as ISO-LATIN-1.
- ▶**UTF_8** Eight-bit Unicode Transformation Format.

8.11.4 Constructor and Destructor Documentation

8.11.4.1 Message(Message& copy)

Copy constructor - initializes the Message object from specified object.

PARAMETERS

- ▶***copy*** Message object to copy.

8.11.5 Member Function Documentation

8.11.5.1 void setRequestID(const std::string& id)

Provides an ID for the playback request. The ID can be any arbitrary string that is unique for the client application in the current session (since the client application calls AudioServer::connect).

Calling this method is optional, and if request ID is not set the system will automatically assign an ID for the request.

PARAMETERS

- ▶***id*** Request ID.

8.11.5.2 std::string getRequestID() const

Returns the ID for the playback request.

RETURNS

- ▶**std::string** Request ID.

8.11.5.3 void setPriority(const MessagePriority& priority)

Specifies the requested message priority. Please refer to the MessagePriority class.

Calling this method is optional, and if message priority is not specified for a request, system defaults will be applied. By default the message will have identical priority to the priority level if the audio source used for the message.

PARAMETERS

- ▶***priority*** Requested message priority.

8.11.5.4 MessagePriority getPriority() const

Returns priority for the current message

RETURNS

- ▶**MessagePriority** Requested message priority.

8.11.5.5 void setPreChime(int chime)

Specifies a chime is to be played prior to the audio content of the message. Chime is specified as a media dictionary item number. Sets audio message contents to a list of dictionary items. Please refer to Section 3.5 Media Dictionary for a definition of dictionary and dictionary items.

PARAMETERS

- ▶***chime*** Dictionary item number of the chime segment in the media dictionary.

8.11.5.6 int getPreChime() const

Returns the pre chime segment number.

RETURNS

- ▶**int** Dictionary item number of the chime segment in the media dictionary.

8.11.5.7 void setPostChime(int chime)

Specifies a chime is to be played after the audio content of the message. Chime is specified as a media dictionary item number. Sets audio message contents to a list of dictionary items. Please refer to Section 3.5 Media Dictionary for a definition of dictionary and dictionary items.

PARAMETERS

- ▶***chime*** Dictionary item number of the chime segment in the media dictionary.

8.11.5.8 int getPostChime() const

Returns the post chime segment number.

RETURNS

- ▶**int** Dictionary item number of the chime segment in the media dictionary.

8.11.5.9 void setGain(double gain)

Specifies the gain offset (in dB) to be applied when playing the audio content of the message. Gain offset is applied to the currently configured device output channel gain for all output channels in the targeted zones.

PARAMETERS

- ▶***gain*** Gain level adjustment, in dB.

8.11.5.10 void setAudioMessageType(Type type)

Sets audio message type to one of the supported formats.

PARAMETERS

- ▶***type*** Message type. All types except for DISPLAY and NONE are accepted as valid arguments to this method.

8.11.5.11 Type Message::getAudioMessageType()

Returns audio message type.

RETURNS

- ▶**Type** Current value of audio source type.

8.11.5.12 void setVisualMessageType(Type type)

Sets visual message type to dictionary items, text or template.

PARAMETERS

- ***type*** Message type. Only DICTIONARY, DISPLAY and TEMPLATE are accepted as valid arguments to this method.

8.11.5.13 Type getVisualMessageType()

Returns visual message type.

RETURNS

- **Type** Current value of visual type.

8.11.5.14 void setAudioMessage(const NumberArray& dictionaryItems)

Sets audio message contents to a list of dictionary items. Please refer to Section 3.5 Media Dictionary for a definition of dictionary and dictionary items. The list of dictionary items will be compared to the latest dictionary loaded on the server. If some of the items in the list do not exist in the dictionary, then an exception will be raised.

This message is valid if audio message type is set to DICTIONARY or NONE; otherwise an exception will be raised. If the audio message type is NONE, it will be set to DICTIONARY once this method is called.

PARAMETERS

- **dictionaryItems** List of dictionary items to announce.

8.11.5.15 void setAudioMessage(const string& pathToRecordedStream, AudioEncoding encoding)

Sets audio message contents to a path to be sent to the NetSpire servers for streaming. The specified path must be available for reading by the application hosting the SDK library.

On message initiation using the method playMessage(), the SDK library will start to transfer the audio content to the NetSpire servers, and the servers will stream the audio content to the recipients on successful transfer. The file will be deleted from the server after announcement completion.

In the event the file cannot be read, or transfer fails, the event PaControllerObserver::onMessageDelete() will be raised to indicate failure with the UUID of the message. Message::setRequestID() must be specified by the client if message tracking is desired in order to track the status of the message.

This message is valid if audio message type is set to RECORDED_STREAM or NONE; otherwise an exception will be raised. If the audio message type is NONE, it will be set to RECORDED_STREAM once this method is called.

PARAMETERS

- **pathToRecordedStream** Path to the file to stream. The file must be a WAV container with the header compliant to RIFF format (i.e. standard WAVE file).
- **encoding** Encoding of audio content within the container. Accepted formats are L16_48K and L16_16K for this method.

8.11.5.16 NumberArray getAudioMessageDictionaryItems() const

Returns list of dictionary items that have been set for playback

RETURNS

- **NumberArray** List of dictionary items to announce.

8.11.5.17 void setAudioMessage(const std::string& text, const std::string& language, const std::string& voice, TextEncoding encoding = UTF_8)

Sets audio message contents to arbitrary text. TTS engine options language, voice / speaker can be specified.

This message is valid if audio message type is set to TTS or NONE; otherwise an exception will be raised. If the audio message type is NONE, it will be set to TTS once this method is called.

PARAMETERS

- **text** Message to announce. An enabled TTS engine generates the audio announcement. Text needs to be encoded as specified in the encoding argument.
- **language** Message language. Supported message languages are determined by TTS licences installed on the audio server or external TTS host. Typical examples of supported languages are “en-us”, “en-uk”, “en-au”, “fr”, “fr-ca”.
- **voice** Message speaker / voice name. Supported voices are determined by TTS licences installed on the audio server or external TTS host. Typical examples of supported languages are “Ava”, and “Amelie”.
- **encoding** Specifies the encoding used for the text argument. Only UTF_8 is supported in the current revision.

8.11.5.18 std::string getAudioMessageTTSText() const

Returns the message which has been set for announcement

RETURNS

- ▶ **std::string** Message to announce.

8.11.5.19 std::string getAudioMessageTTSLanguage() const

Returns the language which has been set for conversion of the message

RETURNS

- ▶ **std::string** Message language.

8.11.5.20 std::string getAudioMessageTTSVoice() const

Returns the voice which has been set for conversion of the message

RETURNS

- ▶ **std::string** Message speaker/voice name.

8.11.5.21 TextEncoding getAudioMessageTTSEncoding() const

Returns the encoding format which has been set to represent the message format.

RETURNS

- ▶ **TextEncoding** Encoding used for text.

8.11.5.22 void setAudioMessage(const std::string& templateName, const AudioServer::KeyValueMap& arguments)

Sets audio message contents in accordance with a pre-defined template. Templates provide a structure for a message. The system substitutes the variables defined in the template with the arguments supplied to this function to create a list of dictionary audio segments to play.

PARAMETERS

- ▶ **templateName** Name of message template to use.
- ▶ **arguments** List of key value pairs to substitute in the template.

8.11.5.23 void setVisualMessage(const NumberArray& dictionaryItems)

Sets visual message contents to a list of dictionary items. Please refer to Section 3.5 Media Dictionary for a definition of dictionary and dictionary items. The list of dictionary items passed to playMessage will be compared to the latest dictionary loaded on the server. If some of the items in the list do not exist in the dictionary, then an exception will be raised.

This message is valid if visual message type is set to DISPLAY or NONE; otherwise an exception will be raised. If the visual message type is NONE, it will be set to DISPLAY once this method is called.

PARAMETERS

- **dictionaryItems** List of dictionary items to display.

8.11.5.24 NumberArray getVisualMessageDictionaryItems() const

Returns list of dictionary items that have been set for playback

RETURNS

- **NumberArray** List of dictionary items to display.

8.11.5.25 void setVisualMessage(const std::string& text)

Sets visual message contents to a specified screen template. The “text” argument contains a screen template defined in accordance with the Netspire XML DCI Specification.

This message is valid if visual message type is set to DISPLAY or NONE; otherwise an exception will be raised. If the visual message type is NONE, it will be set to DISPLAY once this method is called.

PARAMETERS

text Message to display.

8.11.5.26 std::string getVisualMessageText() const

Returns message set for display

RETURNS

- **std::string** Message to display.

8.11.5.27 void setVisualMessage(const std::string& templateName, const AudioServer::KeyValueMap& arguments)

Sets visual message contents in accordance with a pre-defined template. Templates provide a structure for a message and can also include formatting information including screen positions, and visual effects such as scrolling and flashing. The system substitutes the variables defined in the template with the values supplied to this function in rendering the screen contents.

PARAMETERS

- **templateName** Name of message template to use. This argument, when left blank, will not directly affect the currently displayed screen template.

- ▶ **arguments** List of key value pairs to substitute in the template.

8.11.5.28 std::string retrieveAudioMessage()

Initiates retrieval of audio message contents by the API. When the Audio Server processes the request, the method PAControllerObserver::onAudioMessageRetrievalComplete() will be invoked in the application using the API. The observer method can be used to check if the request has been successful, and a HTTP URI to access the audio content.

This method is valid for Messages where content type is TTS. Note for TTS messages, the audio content will be available once the audio is generated using Text to Speech. The Audio Server may elect to re-use previously generated audio content if the audio content for the same TTS message is available from a previous invocation; in which case the audio content will be made available without requiring regeneration.

RETURNS

- ▶ **requestID** Message request ID. This request will be included as part of the arguments to the observer method.

8.12 MessagePriority Class Reference

MessagePriority class includes data members that allow specifying how a message is prioritised in relation to other messages. It is used as a parameter when specifying a message priority using Message::setPriority().

8.12.1 Public Types

- enum **Mode**
  - ABSOLUTE_PRIORITY= 0,
  - RELATIVE_PRIORITY

8.12.2 Enumerated Types

8.12.2.1 enum Mode

Specifies the source type for an audio or visual message.

Enumeration

▶**ABSOLUTE_PRIORITY** Specified message priority level is an absolute value and is not in relation to the priority level of audio source or announcement type

▶**RELATIVE_PRIORITY** Specified message priority level is relative to the priority level of audio source. The priority level of the message is added to the priority level of audio source as an offset when this mode is specified.

8.12.3 Detailed Description

MessagePriority class includes data members that allow specifying how a message is prioritised with other messages. It is used as a parameter when specifying a message priority using Message::setPriority().

8.12.4 Constructor and Destructor Documentation

8.12.4.1 MessagePriority ()

Default constructor - initializes the MessagePriority object with system default priority (RELATIVE_PRIORITY, level 0).

8.12.4.2 MessagePriority (MessagePriority & copy)

Copy constructor - initializes the MessagePriority object from specified object.

PARAMETERS

▶ ***copy*** MessagePriority object to copy.

8.12.4.3 MessagePriority (Mode mode, int level)

Initializes the MessagePriority object using specified priority level and mode.

PARAMETERS

▶ ***mode*** Announcement priority mode.

▶ ***level*** Announcement priority level.

8.12.5 Public Attributes

- *Mode* ***mode***
  Announcement priority mode.

- *int* ***level***
  Priority level. Valid values for ABSOLUTE_PRIORITY is [0, 1000]. Lower numbers indicate lower message priority.

  When RELATIVE_PRIORITY mode is used, the level specified is added to the PaSource priority to determine the final announcement priority. This allows specifying the relative level as 0 and using source priorities as configured during system setup.

  When priority levels are evaluated, if the announcement priority is < 0, it will be increased to 0. If the announcement priority is > 1000, it will be decreased to 1000.

8.12.6 Member Function Documentation

MessagePriority has no member functions.

8.13 ScheduleDefinition Class Reference

ScheduleDefinition class stores information on a scheduled announcement. Announcement schedules can be managed using the methods PAController::createSchedule(), PAController::listSchedules(), and PAController::deleteSchedule().

8.13.1 Public Attributes

- int **Id**
  Id of the schedule message.
- std::string **description**
  Description about this schedule.
- int **startYear**
  Year in which the schedule messages are to start.
- int **startMonth**
  Month in which the schedule messages are to start.
- int **startDay**
  Day in which the schedule messages are to start.
- int **startHour**
  Time in Hours when the schedule messages are to start on each day.
- int **startMinute**
  Time in Minutes when the schedule messages are to start on each day.
- int **endYear**
  Year in which the schedule messages are to finish.
- int **endMonth**
  Month in which the schedule messages are to finish.
- int **endDay**
  Day in which the schedule messages are to finish.
- int **endHour**
  Time in Hours when the schedule messages are to finish on each day.
- int **endMinute**
  Time in Minutes when the schedule messages are to start on each day.
- int **dayMask**
- int **frequency**
  Frequency in seconds to play the message. If field null, frequency will be sent to once per day (i.e. 24 hours).
- std::vector<std::string> **paZones**
  List of target destinations where the message is to be played.
- AnnouncementPriorityMode **priorityMode**

  The priority mode to use when playing the message
- int **priorityLevel**

The priority level (absolute or relative depending on mode) to use when playing the announcement.

- AnnouncementType **announcementType**

  Indicates the type of the announcement. Supported types are AT_NONE, AT_DVA, AT_TTS, AT_TTS_SAF, AT_STORED_DATA, AT_STORED_DATA_SAF. Will be set to AT_NONE if the schedule does not contain an audio component.

- AnnouncementType **visualMessageType**

  Indicates the type of the announcement. Supported types are AT_NONE, AT_DISPLAY_TEMPLATE. Will be set to AT_NONE if the schedule does not contain a visual component.

- std::vector<unsigned int> **dictionaryItems**
  List of dictionary items to be played
- int **repeatCnt**
  Indicates how many times the announcement should be repeated. Applicable if announcement type is AT_DVA
- std:string **ttsLanguage**

  This tells the TTS engine the language to be used. Applicable if announcement type is AT_TTS or AT_TTS_SAF
- std::string **ttsVoice**
  This tells the TTS engine the voice to be used. Applicable if announcement type is AT_TTS or AT_TTS_SAF
- std::string **ttsText**
  This tells the TTS engine the text that needs to be converted for playback. Applicable if announcement type is AT_TTS or AT_TTS_SAF
- int **storedDataSourceId**
  Indicates the ID of the source which which will stream the data. Applicable if announcement type is AT_STORED_DATA or AT_STORED_DATA_SAF
- std::string **storedDataFilePath**
  Indicates the full path of where the media is stored. Applicable if announcement type is AT_STORED_DATA or AT_STORED_DATA_SAF
- std::string **displayTemplateName**
  Indicates the name of the template that needs to be displayed. Applicable if announcement type is AT_DISPLAY_TEMPLATE
- std::map<std::string, std::string> **displayTemplateKeyValueList**
  Indicates a list of variable name and value that needs to be set for the display device. Applicable if announcement type is AT_DISPLAY_TEMPLATE
- int **displayTemplateValidityPeriodSeconds**
  Indicates life of the message. Applicable if the announcement type is AT_DISPLAY_TEMPLATE. Acceptable values are:

  -1: valid forever (message never gets deleted from the queue)

  0 – maxint: validity period in seconds. The message will be attembed to be displayed as soon as the message becomes the highest priority item. The message will stay in the announcement *queue for the specified duration. Note that validity period of 0 indicates the message is “one-shot”. If it can be displayed now, will be displayed, if not will be discarded.*

- double **gain**
  Indicates the value by which the output level needs to be attenuated by. Applicable if the announcement type is AT_DVA
- std::string **chime**
  Indicates the dictionary item number of the chime that will be played at start of the announcement.

8.13.2 Detailed Description

ScheduleDefinition contains information on a scheduled announcement. Scheduled announcements are executed on detinations specified in the schedule by the Communications Exchange Server system.

8.13.3 Member Data Documentation

8.13.3.1 int ScheduleDefinition::dayMask

Days of the week on which the schedule messages are to occur. This field also allows Public Holiday days to be specified. If field is null, Day Mask will default to all days.

Type	Hex	Decimal
None	0	0
Sunday	0x1	1
Monday	0x2	2
Tuesday	0x4	4
Wednesday	0x8	8
Thursday	0x10	16
Friday	0x20	32
Saturday	0x40	64
Public Holiday	0x80	128

8.13.4 Member Function Documentation

ScheduleDefinition has no member functions.

8.14 Alarm Class Reference

Alarm class stores information on an alarm present in the system. In order to get all alarms from the active server, the alarm management needs to be enabled. This is achieved by calling the function AudoServer::enableAlarmManagement(). The function AudioServer::getAlarms() can then be used to get all the alarms present in the active server. Alarm Management can be disabled by calling the function AudioServer::disableAlarmManagement().

8.14.1 Public Attributes

- int **key**
  Key is the unique Id of the Alarm.
- int **almNo**
  current number assigned to this Alarm.
- std::string **almDstName**
  name of device from which the alarm was generated
- std::string **almFirstUpdateTime**
  indicates the date and time when the alarm was first generated. Format is dd-MM-YYYY HH:MM:ss
- std::string **almLastUpdateTime**
  indicates the date and when the the alarm was last generated. Format is dd-MM-YYYY HH:MM:ss
- int **almRepeatCount**
  number of times the alarm has been generated.
- bool **isAlmStateful**
  indicates if the system maintains the state of the alarm.
- bool **almCurrentState**
  indicates if the alarm is active or not.
- int **almLevel**
  indicates the level assigned to the alarm. Level can be INFO, MINOR, MAJOR, SEVERE
- bool **almProtected**
  indicates if the alarm is protected. A protected alarm is not deleted when the system has reached maximum number of configured alarms.
- int **almErrorNo**
  error number assigned to the alarm
- std::string **almErrorText**
  textual description to indicate the reason of number of times the alarm has been generated.
- bool **isAlmAcknowledged**
  indicates if the alarm has been acknowledged by the system operator
- std::string **almAckTimeStamp**
  indicates the date and time when the alarm was acknowledged. Format is dd-MM-YYYY HH:MM:ss
- std::string **almAckUserEndPoint**

  indicates the IP address of the terminal that acknowledged the alarm.

- std::string **almAckUser**
  indicates the username of the operator that acknowledged the alarm
- std::string **almAckDescription**

  textual description for the acknowledgment.

8.14.2 Detailed Description

8.14.3 Constructor and Destructor Documentation

8.14.3.1 Alarm(AlarmMessage& copy)

Copy constructor - initializes the Alarm object from specified object.

PARAMETERS

- ▶ ***copy*** Alarm object to copy.

8.14.4 Member Function Documentation

8.14.4.1 void acknowledgeAlarm(const std::string& endPt, const std::string& operatorName const std::string &ackDescription) throw (std::invalid_argument)

Allows the operator to acknowledge an alarm. An expection is generated if either of the arguments are not entered.

PARAMETERS

- ▶ ***endPt*** indicates the terminal Id which is acknowledging the alarm. This is usually the IP address of the operator terminal
- ▶ **operatorName** indicates the user identification of the operator
- ▶ **ackDescription** indicates a description indicating the reason for acknowledgement.

9 NetSpire SDK Class Reference - Public Address

9.1 Overview

Provides an interface to control and monitor live Public Address subsystem. The main classes used for controlling and monitoring live Public Address are:

PAController class provides an interface retrieve a list of PA sources (e.g. microphones, analogue or digital audio inputs),PA sinks (audio outputs comprising of one or more amplifiers, analogue or digital audio outputs and speakers) and PA zones. The interface allows monitoring the state of all sources and sinks and routing sources to sinks to make announcements.

Routes are established between abstract sources and sinks. Regardless of the location of these sources and sinks, whether they are within the same device, within a number of devices at the same geographic location, or distributed across multiple locations across a large geographic area, the same API is used to establish routes from audio sources to audio sinks. The API makes use of communications mechanisms supported by the NetSpire system transparently, using internal DSP mixers within individual devices, audio streaming within bridged as well as routed IP networks, wireless communication methods including Train Radio, and legacy third party communication networks and protocols to establish routes and make announcements.

PaSource class represents an audio source. These can range from analogue microphones, digital audio sources such as NetSpire IP microphones, analogue balanced/unbalanced audio inputs, digital streamed audio. The client application using the API is required to bind to a PaSource using attachPaSource() if it intends to control PA announcements made from the source. Once the application is bound to a source, it can route the sinks (zones) to be targeted by an announcement from the source, and start/stop PA announcements from the source.

PaSink class represents a target for PA and DVA. Each sink is comprised of one or more amplifier outputs and speakers. PaSink is the smallest audio output grouping that announcements can be directed to. These can represent an audio zone, or can be grouped to create audio zones.

VisualDisplay class represents a display device for displaying messages.

PaZone class represents an audio zone, and is a collection of PaSinks. One PaSink can be a member of zero, one or more zones.

PaTrigger is a software or hardware trigger that indicates the PaSource is now activated. When the trigger is active, PaSource will stream audio to connected PaSinks. Examples of triggers are on-screen Talk buttons, physical PTT buttons on microphones, digital inputs.

PaSelector is a software or hardware zone selector switch. When activated, the audio source to zone routing is changed to reflect the selector position.

9.2 PAController Class Reference

PAController class provides an interface retrieve a list of PA sources (e.g. microphones) PA sinks (audio outputs comprising one or more amplifiers and speakers) and zones. The interface allows monitoring the state of all sources and sinks and making live PA using these components.

9.2.1 Public Types

- typedef std::vector< PaSource > **PaSourceArray**
- typedef std::vector< PaSink > **PaSinkArray**
- typedef std::vector< PaZone > **PaZoneArray**
- typedef std::vector< PaTrigger > **PaTriggerArray**
- typedef std::vector< PaSelector > **PaSelectorArray**

9.2.2 Detailed Description

Provides an interface to control and monitor Public Address subsystem. The PaController can be used to retrieve a list of PA sources (e.g. microphones) and PA sinks (e.g. amplifiers and speakers). The interface allows monitoring activity and making live PA and DVA using these components.

9.2.3 Member Typedef Documentation

- typedef std::vector<PaSource> PAController::**PaSourceArray**
  A collection of PA Sources
- typedef std::vector<PaSink> PAController::**PaSinkArray**
  A collection of PA Sinks
- typedef std::vector<PaZone> PAController::**PaZoneArray**
  A collection of PA Zones
- typedef std::vector<PaTrigger> PAController::**PaTriggerArray**
  A collection of PA Triggers (both hardware and software)
- typedef std::vector<PaSelector> PAController::**PaSelectorArray**
  A collection of PA Zone selectors
- typedef std::vector< ScheduleDefinition > PAController:: **ScheduleDefinitionArray**
  A collection of schedule messages currently stored in the server

9.2.4 Member Function Documentation

9.2.4.1 PAController::PaSourceArray getPaSources()

Returns the list of available PA sources (accessible to this user). This list may be cross checked against a pre-configured source which the program has configured or alternatively provides a list for the user to select one.

RETURNS

- ▶List of available audio sources. The returned list will be empty if not connected to the NetSpire server.

9.2.4.2 bool attachPaSource(int sourceId) throw (std::invalid_argument, std::out_of_range)

Allows an application to bind to a source. Only one application can bind to a source at a time.

PARAMETERS

- ▶***sourceId*** Specifies the selected PA Source Id

RETURNS

- ▶true if binding to source was successful else false

EXCEPTIONS

- ▶***std::invalid_argument*** (IllegalArgumentException in Java) if audio source is not available (due to comms status or is already associated with another application)
- ▶***std::out_of_range*** (IndexOutOf-BoundsException in Java) if audio source Id does not exist.

9.2.4.3 bool detachPaSource(int sourceId) throw (std::invalid_argument, std::out_of_range)

Allows an application to unbind from a source.

PARAMETERS

- ▶***sourceId*** Specifies the selected PA Source Id

RETURNS

- ▶true if unbinding to source was successful else false

EXCEPTIONS

- ▶***std::invalid_argument*** (IllegalArgumentException in Java) if audio source is not available (due to comms status or is already associated with another application)
- ▶***std::out_of_range*** (IndexOutOf-BoundsException in Java) if audio source Id does not exist.

9.2.4.4 PAController::PaSinkArray getPaSinks()

This function returns a list of all audio sinks that can be used to direct announcements. All PaSinks will be listed irrespective of PaSources.

Note that a specific sink can be a fine a granularity as a single speaker and not necessarily refer to a larger area (eg. Factory floor or concourse). The sinkID and descriptions are non-volatile and will be consistent from one application invocation to the next as long as the configuration of the system has not changed. This list will contain all available sinks including those that are not attached to the specified source.

PARAMETERS

- ***sourceId*** Specifies the PA Source Id for which available sinks are being queried.

RETURNS

- List of available audio sinks. The returned list will be empty if not connected to the NetSpire server.

9.2.4.5 PAController::PaSinkArray getPaSinks(int sourceId) throw (std::out_of_range)

This function returns a list of all audio sinks that are available to make announcements to from this source. Note that a specific sink can be a fine a granularity as a single speaker and not necessarily refer to a larger area (eg. Factory floor or concourse). The sinkID and descriptions are non-volatile and will be consistent from one application invocation to the next as long as the configuration of the system has not changed. This list will contain all available sinks including those that are not attached to the specified source.

PARAMETERS

- ***sourceId*** Specifies the PA Source Id for which available sinks are being queried.

RETURNS

- List of available audio sinks. The returned list will be empty if not connected to the NetSpire server.

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBoundsException in Java) if sourceId does not exist.

9.2.4.6 StringArray getPaZones()

This function returns a list of all PaZones defined in the system. Zones are part of the system configuration and can be customised by users to suit system zoning requirements.

Note that a zone can represent a fine a granularity such as a single speaker and as well as a larger area (eg. Factory floor, concourse, station, rail corridor). The Zone ID and descriptions are non-volatile and will be consistent from one application invocation to the next as long as the configuration of the system has not changed. This list will contain all available zones.

RETURNS

▶ List of the names of available audio zones. The returned list will be empty if not connected to the NetSpire server.

9.2.4.7 PaSink::State getPaSinkState(int sinkId) throw (std::out_of_range)

Allows an application to get the current state of the sink

PARAMETERS

▶ ***sinkId*** Specifies the current sink

RETURNS

▶ ***PaSink::State*** Returns the current state of the sink

EXCEPTIONS

▶ ***std::out_of_range*** (IndexOutOfBoundsException in Java) if audio sink Id does not exist.

9.2.4.8 void setPaMonitorSink(int sinkId, int sinkIdToMonitor) throw (std::out_of_range)

Allows an application to monitor a particular sink on another sink.

PARAMETERS

▶ ***sinkId*** Specifies the sink that will monitor

▶ ***sinkIdToMonitor*** Specifies the sink that will be monitored. -1 cancels/disables monitoring for sinkId.

EXCEPTIONS

▶ ***std::out_of_range*** (IndexOutOfBoundsException in Java) if audio sink Id does not exist.

9.2.4.9 PAController::PaTriggerArray getHwPaTriggers(int sourceId) throw (std::out_of_range, std::invalid_argument)

Returns a list of available hardware triggers for a particular source Id. A PA announcement will start when a trigger is activated and end when the trigger is de-activated.

PARAMETERS

▶ ***sourceId*** Specifies the selected PA Source Id

RETURNS

▶ Returns a list of available hardware triggers for the selected PA Source Id

EXCEPTIONS

- ***std::invalid_argument*** (IllegalArgumentException in Java) if audio source is not available (due to comms status or is already associated with another application)
- ***std::out_of_range*** (IndexOutOf-BoundsException in Java) if audio source Id does not exist.

9.2.4.10 PaTrigger::State getHwPaTriggerState(int triggerId) throw (std::out_of_range)

Allows an application to get the current state of the hardware trigger.

PARAMETERS

- ***triggerId*** Specifies the selected PA Trigger Id

RETURNS

- Returns the current state of the selected PA Trigger Id

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBoundsException in Java) if trigger Id does not exist.

9.2.4.11 int createSwPaTrigger(int sourceId, int triggerPriority) throw (std::out_of_range, std::invalid_argument)

Allows a software trigger to be created by the application. This allows a GUI application or third party interface or hardware device to make announcements in the system.

PARAMETERS

- ***sourceId*** Specifies the selected PA Source Id
- ***triggerPriority*** The priority level for the new SW trigger. Value is in the range of 0 being lowest and 254 being highest.

RETURNS

- ***triggerId*** Returns the newly created Trigger Id

EXCEPTIONS

- ***std::invalid_argument*** (IllegalArgumentException in Java) if audio source Id is not available (due to comms status or is already associated with another application)
- ***std::out_of_range*** (IndexOut-OfBoundsException in Java) if audio source Id does not exist.

- ▶ ***std::out_of_range*** (IndexOutOf-BoundsException in Java) if triggerPriority is outside 0-254

9.2.4.12 void deleteSwPaTrigger(int triggerId) throw (std::out_of_range)

Allows a software trigger that has been previously created by application to be deleted.

PARAMETERS

- ▶ ***triggerId*** Specifies the selected Trigger Id

EXCEPTIONS

- ▶ ***std::out_of_range*** (IndexOutOfBoundsException in Java) if trigger Id does not exist.

9.2.4.13 void activateSwPaTrigger(int triggerId) throw (std::out_of_range)

Allows a software trigger that has been previously created to be activated. Note that hardware triggers cannot be triggered via this function call. A typical use for this function would be to bind it to the button down event for a PTT button on a GUI console.

PARAMETERS

- ▶ ***triggerId*** Specifies the selected Trigger Id

EXCEPTIONS

- ▶ ***std::out_of_range*** (IndexOutOfBoundsException in Java) if trigger Id does not exist.

9.2.4.14 void deactivateSwPaTrigger(int triggerId) throw (std::out_of_range)

Allows a software trigger that has been previously activated to be de-activated. Note that hardware triggers cannot be de-activated via this function call. A typical use for this function would be to bind it to the button up event for a PTT button on a GUI console.

PARAMETERS

- ▶ ***triggerId*** Specifies the selected Trigger Id

EXCEPTIONS

- ▶ ***std::out_of_range*** (IndexOutOfBoundsException in Java) if trigger Id does not exist.

9.2.4.15 PAController::PaSelectorArray getPaSelectors(int sourceId) throw (std::out_of_range)

Returns a list of all available zone selectors for a particular source. Zone selectors is a mechanism via which hardware inputs can be configured to associated and activate sinks on a source. Zone selectors can be configured in the system to be non-volatile and non-modifiable or can be configured to be under software control.

PARAMETERS

- ***sourceId*** Id of the PA source for which to list the zone selectors

RETURNS

- List of available zone selectors. The returned list will be empty if not connected to the NetSpire server.

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBoundsException in Java) if source Id does not exist.

9.2.4.16 PAController::PaZoneArray getPaZonesForSelector(int selectorId) throw (std::out_of_range)

Allows sinks associated with a zone selector to be retrieved.

PARAMETERS

- ***selectorId*** Id of the zone selector whose associated sinks need to be listed

RETURNS

- List of available sinks. The returned list will be empty if not connected to the Netspire server.

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBoundsException in Java) if selecotrId does not exist

9.2.4.17 void addPaZoneToSelector(int selectorId, const std::string& zoneId) throw (std::out_of_range)

Adds a sink to software controllable so that when this selector is activated, the sink will be included in the announcement area.

PARAMETERS

- ***selectorId*** Id of the zone selector to which the sink will be added
- ***zoneId*** Id of the zone to be included in the announcement area

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBounds in Java) if selectorId or sinkId does not exist

9.2.4.18 void deletePaZoneFromSelector(int selectorId, const std::string& zoneId) throw (std::out_of_range)

Removes a specific sink form a software controllable zone selector or for all sinks to be deleted from the software controllable zone selector.

PARAMETERS

- ***selectorId*** Id of the zone selector from which sink(s) will be removed
- ***zoneId*** Id of the zone to be removed from the zone selector. Blank indicates all zones to be removed.

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBounds in Java) if selectorId or sinkId does not exist

9.2.4.19 void enablePaSelector(int selectorId) throw (std::out_of_range)

Allows the software to enable a zone selector

PARAMETERS

- ***selectorId*** Id of the zone selector to be enabled

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBounds in Java) if selectorId does not exist

9.2.4.20 void disablePaSelector(int selectorId) throw (std::out_of_range)

Allows the software to disable a zone selector

PARAMETERS

- ***selectorId*** Id of the zone selector to be disabled

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBounds in Java) if selectorId does not exist

9.2.4.21 PaSelector::State getPaSelectorState(int selectorId) throw (std::out_of_range)

Returns the current state of the zone selector.

PARAMETERS

- ***selectorId*** Id of the zone selector whose state needs to be queried

RETURNS

- current state of the zone selector as stated in the enumeration PaSelector state

EXCEPTIONS

- ***std::out_of_range*** (IndexOutOfBounds in Java) if selectorId does not exist

9.2.4.22 void registerObserver(PAControllerObserver *paObserver)

Registers an observer with the PAControllerClass instance. Callback methods on the observer will be called on state changes.

PARAMETERS

- ***paObserver*** Observer instance

9.2.4.23 unsigned int playMessage (const StringArray & zones, const StringArray &visualDevices, const Gain &gain, const NumberArray &dictionaryItems, const char *visualText, bool resumeOnInterruptFlag, bool overrideExisting, unsigned int validityPeriod, unsigned int priority, unsigned int mode) throw (std::invalid_argument, std::out_of_range)

Initiate message on Audio Zones and/or Visual Displays. A single function is used to allow queuing and synchronisation of audio and visual information.

PARAMETERS

- ***zones*** List of audio zones in the system to be used for audio output. List of all zones and their names can be retrieved using the method PAController::getPaZones().

- ***visualDevices*** List of visual device names to be used for visual output. List of all visual devices and their names can be retrieved using the method PAController::getVisualDevices().
- ***gain*** The output gain level to be applied to the amplifiers when playing audio
- ***dictionaryItems*** List of dictionary items to be played. Please refer to Section 7.6 Media Dictionary for a definition of dictionary and dictionary items. The list of dictionary items passed to playMessage will be compared to the latest dictionary loaded on the server. If some of the items in the list do not exist in the dictionary, then an exception will be raised.
- ***visualText*** Textual information displayed on LED/LCD passenger information displays specified in the visualDevices argument. A NULL pointer can be sent to indicate the visual components of the dictionary items specified in the dictionaryItems argument should be displayed on visualDevices. An empty string will clear the displays.
- ***resumeOnInterruptFlag*** allows DVA announcements that are interrupted by PA announcements to be restarted from the beginning. This argument is reserved for future use and the current API release only supports "false" as a valid value for this argument. Sending true will raise an invalid_argument exception.
- ***overrideExisting*** When set to true, any currently playing DVA or PA message will be interrupted and the specified DVA message will start playing immediately. Set to false in most instances, which will cause the DVA to be queued to play after the currently queued items. Set to true when the DVA message has to be played immediately e.g. to play a door obstruction message
- ***validityPeriod*** indicates the time in milliseconds of how long a message will be displayed on a display. A value of 0 indicates display message forever or until overwritten by another message.
- ***priority*** This argument is reserved for future use.
- ***mode*** This argument is reserved for future use.

RETURNS

- A message ID which can be used by cancelMessage().

EXCEPTIONS

- ***std::invalid_argument, std::out_of_range*** (In Java, IllegalArgumentException or IndexOutOf-BoundsException respectively)

9.2.4.24 std::string playMessage(const StringArray& zones, Message& message)

Initiate message on Audio Zones and/or Visual Displays. A single function is used to allow queuing and synchronisation of audio and visual information.

PARAMETERS

- ***zones*** List of zones in the system to be used for audio and visual output. List of all zones and their names can be retrieved using the method PAController::getPaZones().
- ***message*** Message to be played at target sinks and devices. Please refer to message class defined in 8.11

RETURNS

- A message ID which can be used by cancelMessage

9.2.4.25 void cancelMessage(unsigned int messageId) throw (std::invalid_argument)

Enables message that has been previously initiated to be cancelled by using the messageId returned by playMessage().

PARAMETERS

- ***messageId*** Identifier for message as returned by playMessage(). Set it to 0 to cancel all messages.

EXCEPTIONS

- ***std::invalid_argument*** (an IllegalArgumentException in Java)

9.2.4.26 void cancelMessage(const std::string& requestId) throw (std::invalid_argument)

Enables message that has been previously initiated to be cancelled by using the messageId returned by playMessage().

PARAMETERS

- ***messageId*** Identifier for message as returned by playMessage().Set it to empty string to cancel all messages.

EXCEPTIONS

- ***std::invalid_argument*** (an IllegalArgumentException in Java)

9.2.4.27 void cancelMessageByPaZone(const StringArray& zones, bool cancelAll) throw (std::invalid_argument)

Enables cancelling current or all messages that being played/queued to be played at a list of audio sinks (zones).

PARAMETERS

- ***zones*** List of audio zones in the system to be used for audio output.
- ***cancelAll*** When set to true, all currently playing/queued messages will be cancelled. Set to false to interrupt and cancel the currently playing message on selected zone(s).

EXCEPTIONS

- ***std::invalid_argument*** (an IllegalArgumentException in Java)

9.2.4.28 void createSchedule(const ScheduleDefinition &scheduleDef) throw (std::invalid_argument, std::out_of_range)

Schedule messages on Audio Zones and or Visual Displays. A single schedule includes both audio and visual messages, which can be useful to queue both types of actions and execute both audio and visual messages at the same time.

Messages can be scheduled to be played in the future, or repeated within specified time and date periods. A schedule can be created where date/time for the schedule is in the past, in this case the system will not execute the action unless the system time is altered e.g. via the web UI or NTP.

PARAMETERS

- ***scheduleDef*** contains all the information to schedule the message.

EXCEPTIONS

- ***std::invalid_argument*** (IllegalArgumentException in Java) if no audio sinks specified
- ***std::out_of_range*** (IndexOutOfBoundsException in Java) if any of the schduleInfo is incorrect

9.2.4.29 PaController::ScheduleDefinitionArray listSchedules()

List all schedule messages present in the AS.

RETURNS

- ScheduleDefinitionArray a list of schedules.

9.2.4.30 void deleteSchedule(int scheduleId) throw (std::invalid_argument, std::out_of_range)

Delete an existing scheduled message from the AS.

PARAMETERS

- ***scheduleId*** id of the schedule to be deleted from the system. The id is included in the ScheduleDefinition class.

EXCEPTIONS

- ***std::invalid_argument*** (IllegalArgumentException in Java) if schdule Id is invalid

- ***std::out_of_range*** (IndexOutOfBoundsException in Java) if audio schdule Id is not in range

9.3 PaSource Class Reference

PaSource class represents an audio source. These can range from analogue microphones, digital audio sources such as NetSpire IP microphones, analogue balanced/unbalanced audio inputs and digital streamed audio. This class is used to retrieve information on the audio source and set its gain level.

9.3.1 Public Types

- enum **Type**
  - SOURCE_TYPE_NOT_DEFINED = 0,
  - NETSPIRE_ANALOG_INPUT,
  - NETSPIRE_IP_PAGING_STATION,
  - PC_AUDIO_INPUT,
  - NETSPIRE_HANDSET,
  - SIP_HANDSET,
  - TRAIN_RADIO,
  - EWIS, BGM_SERVER
- enum **AttachState**
  - UNKNOWN,
  - NOT_ATTACHED,
  - ATTACHED
- enum **AttachMode**
  - ADD_TO_EXISTING_SET = 0,
  - REPLACE_EXISTING_SET
- enum **HealthTestMode**
  - TEST_DISABLED = 0,
  - TEST_ON_DEMAND,
  - TEST_AUTOMATIC
- enum **HealthState**
  - UNKNOWN_HEALTHSTATE = 0,
  - HEALTHY,
  - FAULTY

9.3.2 Public Attributes

- int **id**
  Id of the PA source. This is for the application to use as an Id to request a source. For Example: 66112.

- std::string **location**
  Descriptive name (suitable for display) for the location of the source. For Example: "Domestic Terminal: Control Room 1".
- std::string **label**
  Descriptive name given to a source. For Example: "Primary PA Console".
- std::string **ipAddress**
  Source IP Address (if this information is available).
- Type **type**
  Audio source type. Possible values are listed in the enumeration Type.
- std::string **subAddress**
- State **state**
  Indicates the current state of the source.

9.3.3 Detailed Description

PaSource is used to define an audio source type that can be used in various PA initiation methods.

9.3.4 Constructor and Destructor Documentation

9.3.4.1 PaSource (const PaSource &paSourceAnother)

Creates a duplicate copy of the PaSource object using the copy constructor.

PARAMETERS

- ▶**paSourceAnother** another instance of the PaSource object which needs to be duplicated for internal storage

9.3.5 Member Enumeration Documentation

9.3.5.1 enum PaSource::Type

Audio Source Type

ENUMERATION

- ▶**SOURCE_TYPE_NOT_DEFINED** Third-party source
- ▶**NETSPIRE_ANALOG_INPUT** An Analogue Audio Input connected to a NetSpire device such as Network Audio Controller
- ▶**NETSPIRE_IP_PAGING_STATION** IP based Microphone
- ▶**PC_AUDIO_INPUT** Microphone connected to PC Input

- ▶ **NETSPIRE_HANDSET** SIP Source
- ▶ **SIP_HANDSET** Voice Over IP Input
- ▶ **TRAIN_RADIO** Train Radio Input
- ▶ **EWIS** Emergency Input
- ▶ **BGM_SERVER** Background Music

9.3.5.2 enum PaSource::HealthTestMode

Audio Source health test mode.

ENUMERATION

- ▶ **TEST_DISABLED** Health tests disabled – no health tests are supported for the source
- ▶ **TEST_ON_DEMAND** Health tests are enabled, and require manual or API initiation
- ▶ **TEST_AUTOMATIC** Health tests are enabled, and conducted automatically (either continuous or periodic)

9.3.5.3 enum PaSource::HealthState

Health State

ENUMERATION

- ▶ **UNKNOWN_HEALTHSTATE** The device is not setup for health check and cannot report a health status. Please note some device types do not support health monitoring and will always this state.
- ▶ **HEALTHY** The source is detected to be healthy.
- ▶ **FAULTY** A fault is detected with the source e.g. microphone detached or capsule failure.

9.3.5.4 enum PaSource::AttachState

Audio Source attachement status.

ENUMERATION

- ▶ **UNKNOWN** Source state unknown
- ▶ **NOT_ATTACHED** Source is not currently attached to the system
- ▶ **ATTACHED** Source is attached to the system

9.3.6 Member Data Documentation

9.3.6.1 std::string PaSource::subAddress

The Audio Source Sub-Address provides a device specific sub-address. For Example:

- ▶**NetSpire Analog Input** Analog Port Number
- ▶**NetSpire IP Paging Station** 2
- ▶**PC Audio Input** VCI port number
- ▶**NetSpire Handset** Handset SIP Phone Number
- ▶**SIP Handset** Phone Number
- ▶**Train Radio** None (empty string)

9.3.7 Member Function Documentation

9.3.7.1 void setGain (double gain, bool persistence = true, bool delayPersistence = true) throw (std::invalid_argument, std::out_of_range)

Allows controlling the gain level of a PaSource. This function is valid for the following PaSource types:

- ▶NETSPIRE_ANALOG_INPUT
- ▶NETSPIRE_IP_PAGING_STATION

PARAMETERS

- ▶***double*** New desired gain level. Must be a double value between [-96.00, +42.00]
- ▶***bool*** flag to indicate if changes should be persistence. The default behaviour is that all changes will be persistence. Note that there will be a system delay when changes are made persistant.
- ▶***bool*** flag to indicate if settings should be made persistence after a delay. If this argument is set to true, the changes are made persistent only if no gain updates are received for 20 seconds. Everytime a gain update is received; the timer will be extended so the final changes are committed 20 seconds after the last gain update received.

EXCEPTIONS

- ▶C++ std::invalid_argument, std::out_of_range
- ▶Java, IllegalArgumentException, IndexOutOf-BoundsException

If the PaSource is not one of the supported PaSource types or if the gain level is outside the specified range.

9.3.7.2 double getGain() throw (std::invalid_argument)

Returns the gain level of a PaSource. This function is valid for the following PaSource types:

▶NETSPIRE_ANALOG_INPUT

▶NETSPIRE_IP_PAGING_STATION

RETURNS

▶***double*** Gain level of the specified PaSource. The Gain is a double in the range [-96.00, +42.00]

EXCEPTIONS

▶C++ std::invalid_argument

▶Java, IllegalArgumentException

If the PaSource is not one of the supported PaSource types.

9.3.7.3 HealthTestMode getHealthTestMode() throw (std::invalid_argument)

Returns health test mode for a PaSource. This function is valid for the following PaSource types:

▶NETSPIRE_ANALOG_INPUT

▶NETSPIRE_IP_PAGING_STATION

RETURNS

▶***HealthTestMode*** Health test mode of the specified PaSource – valid values are:

- TEST_DISABLED: Tests are disabled for this sink
- TEST_ON_DEMAND: Tests only conducted on user request
- TEST_AUTOMATIC: Tests conducted automatically

EXCEPTIONS

▶C++ std::invalid_argument

▶Java, IllegalArgumentException

If the PaSource is not one of the supported PaSource types.

9.3.7.4 void initiateHealthTest() throw (std::invalid_argument)

Initiates health test for a PaSource. This function is valid for the following PaSource types:

- ▶NETSPIRE_ANALOG_INPUT
- ▶NETSPIRE_IP_PAGING_STATION

The current state of health test can be observed using the getState() method.

EXCEPTIONS

- ▶C++ std::invalid_argument
- ▶Java, IllegalArgumentException

If the PaSource is not one of the supported PaSource types.

9.3.7.5 HealthState getHealthState() const

Returns the health state of a PaSource.

RETURNS

- ▶***HealthState*** Current state of the specified PaSource

9.3.7.6 AttachState getAttachState() const

Returns the status of a PaSource.

RETURNS

- ▶***AttachState*** Current state of the specified PaSource

9.3.7.7 PAController::PaZoneArray getAttachedPaZones()

Returns a list of all audio zones that are currently attached to the source at the time of the call

RETURNS

- ▶Returns a list of PA zones that are currently attached to the PA source

9.3.7.8 void attachPaZone(const std::string& zoneId, AttachMode mode) throw (std::out_of_range, std::invalid_argument)

Allows a zone to be manually attached to a source so that when a trigger for the source is activated, the zone will be included in the output announcement.

PARAMETERS

- ***zoneId*** Specifies the Id of the zone to attach
- ***mode*** identifies whether the zone is to be added to the existing set or replace the existing set.

EXCEPTIONS

- ***std::invalid_argument*** (IllegalArgumentException in Java) if audio source or audio sink is not available (due to comms status or is already associated with another application) ***std::out_of_range*** (IndexOutOfBoundsException in Java) if audio source Id or sink Id does not exist.

9.3.7.9 void detachPaZone(const std::string& zoneId) throw (std::out_of_range, std::invalid_argument)

Allows one zone to be detached from a source so that when a trigger for the source is activated, the zone will not be included in the output announcement.

PARAMETERS

- ***zoneId*** Specifies the PA Zone Id to be detached. .

EXCEPTIONS

- ***std::invalid_argument*** (IllegalArgumentException in Java) if audio source or audio zone is not available (due to comms status or is already associated with another application)

9.3.7.10 void detachAllPaZones() throw (std::invalid_argument)

Allows all attached zones to be detached from a source so that when a trigger for the source is activated, no zones will not be included in the output announcement.

EXCEPTIONS

- ***std::invalid_argument*** (IllegalArgumentException in Java) if audio source is not available (due to comms status or is already associated with another application)

9.4 PaSink Class Reference

PaSink class represents a target for PA and DVA. These form part of audio zones, and each sink is comprised of one or more amplifier outputs and speakers.

This class provides an interface to query the status of a sink, and set their configuration including gain levels.

9.4.1 Public Types

- enum **Type**
  - SINK_TYPE_NOT_DEFINED = 0,
  - HIGH_IMPEDANCE_SPEAKER_BUS,
  - LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT,
  - INDUCTION_LOOP_OUTPUT,
  - LINE_LEVEL_AUDIO_OUTPUT,
  - INTERCOM_HELP_POINT,
  - TELEPHONY_ENDPOINT,
  - RADIO_REBROADCAST
- enum **State**
  - UNKNOWN = 0,
  - IDLE,
  - ACTIVE
- enum **AnnouncementType**
  - NO_ANNOUNCEMENT = 0,
  - FILE_PLAY,
  - LLPA_BUFFERED,
  - LLPA_REAL_TIME,
  - AUDIO_SWITCHING,
  - VOIP_STREAMING
- enum **HealthState**
  - UNKNOWN_HEALTHSTATE = 0,
  - HEALTHY,
  - MINOR_FAULT,
  - MAJOR_FAULT
- enum **HealthTestMode**
  - TEST_DISABLED = 0,
  - TEST_ON_DEMAND,
  - TEST_AUTOMATIC

- enum **HealthTestStatus**
  - TEST_PENDING = 0,
  - TEST_NONE,
  - TEST_IN_PROGRESS

9.4.2 Public Attributes

- int **id**
  Id of the PA sink. This is for the application to use as an Id to request an audio sink. For Example: 10.
- std::string **location**
  Descriptive name for the location of the audio sink. For Example: "Central Station".
- std::string **label**
  A textual description of the sink (suitable for display). For Example: "Platform 1 - Induction Loop".
- std::string **ipAddress**
  Sink IP Address (if this information is available).
- Type **type**
  Audio sink type. Possible values are listed in the enumeration Type.
- std::string **subAddress**
- State **state**
  Indicates the current state of the sink.
- AnnouncementType **announcementType**
  Indicates the currnet announcement type that is playing on the sink.
- std::string **outputGain**
  Indicates the current output gain.
- bool **muteActive**
  Indicates if the sink is muted or not.
- int **monitoringSinkId**
  Indicates the sink Id that is currently monitored by this sink.
- HealthState **healthState**
  Indicates the health state of the sink.

9.4.3 Detailed Description

PaSink is used to define an audio sink to make announcement to from a source

9.4.4 Constructor and Destructor Documentation

9.4.4.1 PaSink (const PaSink& paSinkAnother)

Creates a duplicate copy of the PaSink object using the copy constructor.

PARAMETERS

- ▶**paSinkAnother** another instance of the PaSink object which needs to be duplicated for internal storage.

9.4.5 Member Enumeration Documentation

9.4.5.1 enum PaSink::Type

Sink Type

ENUMERATION

- ▶**SINK_TYPE_NOT_DEFINED** Third-Party sink (not NetSpire device)
- ▶**HIGH_IMPEDANCE_SPEAKER_BUS** NetSpire high impedance speaker output - generally 100V Audio Line
- ▶**LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT** NetSpire low impedance speaker output - generally 2 ohm Audio
- ▶**INDUCTION_LOOP_OUTPUT** NetSpire induction loop output or HIILS
- ▶**LINE_LEVEL_AUDIO_OUTPUT** NetSpire line level audio which can be fed in 3rd party sinks
- ▶**INTERCOM_HELP_POINT** Reserverd for future implementation
- ▶**TELEPHONY_ENDPOINT** Reserverd for future implementation
- ▶**RADIO_REBROADCAST** Reserverd for future implementation

9.4.5.2 enum PaSink::Mode

Sink Mode Type

ENUMERATION

- ▶**ADD_TO_EXISTING_SET** Allows sinks to be added to existing set using software
- ▶**REPLACE_EXISTING_SET** Replaces existing set of sinks

9.4.5.3 enum PaSink::State

Sink activity state.

ENUMERATION

- ▶**UNKNOWN** Sink state unknown

- ▶**IDLE** Sink is Idle
- ▶**ACTIVE** Sink is busy playing stream

9.4.5.4 enum PaSink::AnnouncementType

Announcement Type

ENUMERATION

- ▶**NO_ANNOUNCEMENT** PaSink is idle (i.e. no audio activity)
- ▶**FILE_PLAY** Playing a pre-recorded announcement; typically this a DVA message
- ▶**LLPA_BUFFERED** Buffered Long Line Public Announcement. Announcement is buffered before playback.
- ▶**LLPA_REAL_TIME** Real-time Long Line Public Announcement. Announcement is played in real-time.
- ▶**AUDIO_SWITCHING** PaSink is playing an audio fed through NetSpire device
- ▶**VOIP_STREAMING** PaSink is playing a Public Announcement (PA). Typically this is an audio stream from a PaSource.

9.4.5.5 enum PaSink::HealthState

Health State

ENUMERATION

- ▶**UNKNOWN_HEALTHSTATE** The device is not setup for speaker health check and cannot report a health status. Please note some device types such as some device's preview speakers don't support health monitoring and will always this state.
- ▶**HEALTHY** All speakers on the bus are reported to be within the calibrated value.
- ▶**MINOR_FAULT** One or some but not all of the speakers on a multi-speaker bus is reported as faulty.
- ▶**MAJOR_FAULT** The system is reporting a total outage for the PaSink or a significant percentage of the speakers on a multi-speaker bus are reported as faulty.

9.4.5.6 enum PaSink::HealthTestMode

Audio Sink health test mode.

ENUMERATION

- ▶**TEST_DISABLED** Health tests disabled – no health tests are supported for the sink

- **TEST_ON_DEMAND** Health tests are enabled, and require manual or API initiation
- **TEST_AUTOMATIC** Health tests are enabled, and conducted automatically (either continuous or periodic)

9.4.6 Member Data Documentation

9.4.6.1 std::string PaSink::subAddress

Provides a device specific sub-address. For Example:

- High Impedance Speaker Bus Amplifier Port Number
- Line Level Audio Output VCI Port Number

9.4.7 Member Function Documentation

9.4.7.1 void PaSink::setGain (Gain gain, bool persistence = true, bool delayPersistence = true) throw (std::invalid_argument, std::out_of_range)

Allows controlling the gain level of a PaSink. This function is valid for the following PaSink types:

- HIGH_IMPEDANCE_SPEAKER_BUS
- LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT
- INDUCTION_LOOP_OUTPUT
- LINE_LEVEL_AUDIO_OUTPUT

PARAMETERS

- ***Gain*** New desired gain level. The Gain class allows a gain range of [-96.000, 0.000]
- ***bool*** flag to indicate if changes should be persistent. The default behaviour is that all changes will be persistent. Note that there will be a system delay when changes are made persistant.
- ***bool*** flag to indicate if settings should be made persistence after a delay. If this argument is set to true, the changes are made persistent only if no gain updates are received for 20 seconds. Everytime a gain update is received; the timer will be extended so the final changes are committed 20 seconds after the last gain update received.

EXCEPTIONS

- C++ std::invalid_argument, std::out_of_range
- Java, IllegalArgumentException, IndexOutOf-BoundsException

If the PaSink is not one of the supported PaSink types or if the gain level is outside the specified range.

9.4.7.2 Gain PaSink::getGain() throw (std::invalid_argument, std::out_of_range)

Returns the gain level of a PaSink. This function is valid for the following PaSink types:

- ▶HIGH_IMPEDANCE_SPEAKER_BUS
- ▶LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT
- ▶INDUCTION_LOOP_OUTPUT
- ▶LINE_LEVEL_AUDIO_OUTPUT

RETURNS

- ▶***Gain*** Gain level of the specified PaSink. The Gain class stores the gain in the range of [-96.000, 0.000]

EXCEPTIONS

- ▶C++ std::invalid_argument
- ▶Java, IllegalArgumentException

If the PaSink is not one of the supported PaSink types.

9.4.7.3 void PaSink::mute()

Mutes audio output at the audio sink.

9.4.7.4 void PaSink::unmute()

Unmutes audio output at the audio sink.

9.4.7.5 bool PaSink::isMuted() const

Queries audio output mute status at the audio sink.

RETURNS

- ▶***bool*** True if sink is muted, false if not muted

9.4.7.6 HealthTestMode PaSink::getHealthTestMode() throw (std::invalid_argument)

Returns health test mode for a PaSink. This function is valid for the following PaSink types:

- ▶HIGH_IMPEDANCE_SPEAKER_BUS

▶LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT

▶INDUCTION_LOOP_OUTPUT,

▶INTERCOM_HELP_POINT

RETURNS

▶***HealthTestMode*** Health test mode of the specified PaSink – valid values are:

- TEST_DISABLED: Tests are disabled for this sink
- TEST_ON_DEMAND: Tests only conducted on user request
- TEST_AUTOMATIC: Tests conducted automatically

EXCEPTIONS

▶C++ std::invalid_argument

▶Java, IllegalArgumentException

If the PaSink is not one of the supported PaSink types.

9.4.7.7 void PaSink::initiateHealthTest() throw (std::invalid_argument)

Initiates health test for a PaSink. This function is valid for the following PaSink types:

▶HIGH_IMPEDANCE_SPEAKER_BUS

▶LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT

▶INDUCTION_LOOP_OUTPUT,

▶INTERCOM_HELP_POINT

The current state of health test can be observed using the getHealthTestStatus() method.

EXCEPTIONS

▶C++ std::invalid_argument

▶Java, IllegalArgumentException

If the PaSink is not one of the supported PaSink types.

9.4.7.8 HealthTestStatus PaSink::getHealthTestStatus() const

Returns the current health test state for a PaSink. This function is valid for the following PaSink types:

▶HIGH_IMPEDANCE_SPEAKER_BUS

- ▶LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT
- ▶INDUCTION_LOOP_OUTPUT,
- ▶INTERCOM_HELP_POINT

RETURNS

▶***HealthTestStatus*** Health test status of the specified PaSink – valid values are:

- TEST_PENDING: PA Sink is currently not Idle, and will be tested when next idle.
- TEST_NONE: PA Sink is not being tested.
- TEST_IN_PROGRESS: PA Sink testing in progress. Once the state changes from TEST_IN_PROGRESS to TEST_NONE, the property healthState will be updated to indicate the test result.

9.4.7.9 void PaSink::setPAMonitoring(bool enabled) throw (std::invalid_argument)

Enables monitoring PA and DVA announcements on the sink on supported monitoring devices. This function is valid for the following PaSink types:

- ▶HIGH_IMPEDANCE_SPEAKER_BUS
- ▶LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT
- ▶INDUCTION_LOOP_OUTPUT
- ▶LINE_LEVEL_AUDIO_OUTPUT

PARAMETERS

▶***enabled*** If set to true, PA/DVA announcements will be monitored on the sink. If set to false PA/DVA announcements will not be monitored.

EXCEPTIONS

- ▶C++ std::invalid_argument
- ▶Java, IllegalArgumentException

If the PaSink is not one of the supported PaSink types.

9.4.7.10 bool PaSink::getPAMonitoring() throw (std::invalid_argument)

Returns the current state of PA and DVA announcement monitoring on the sink. This function is valid for the following PaSink types:

- ▶HIGH_IMPEDANCE_SPEAKER_BUS
- ▶LOW_IMPEDANCE_AMPLIFIER_SPEAKER_OUTPUT

- ▶INDUCTION_LOOP_OUTPUT
- ▶LINE_LEVEL_AUDIO_OUTPUT

RETURNS

- ▶True if PA/DVA announcements are monitored on the sink. False if PA/DVA announcements are not monitored.

EXCEPTIONS

- ▶C++ std::invalid_argument
- ▶Java, IllegalArgumentException

If the PaSink is not one of the supported PaSink types.

9.5 VisualDisplay Class Reference

VisualDisplay class represents a display device suitable for displaying visual output such as static or dynamic text, images, videos, and customised display templates including these visual types.

This class provides an interface to query the presence and status of visual devices in the system.

9.5.1 Public Types

- enum **CommsType**
  - SERIAL = 0,
  - IP_V4,

    IP_V6
- enum **HealthState**
  - UNKNOWN = 0,
  - HEALTHY,

    FAULTY
- enum **HealthTestMode**
  - TEST_DISABLED = 0,
  - TEST_ON_DEMAND,
  - TEST_AUTOMATIC
- enum **HealthTestStatus**
  - TEST_PENDING = 0,
  - TEST_NONE,
  - TEST_IN_PROGRESS

9.5.2 Public Attributes

- int **id**
  Id of the PA sink. This is for the application to use as an Id to request an audio sink. For Example: 10.
- std::string **location**
  Descriptive name for the location of the audio sink. For Example: "Central Station".
- std::string **label**
  A textual description of the sink (suitable for display). For Example: "Platform 1 - Induction Loop".
- std::string **ipAddress**
  Sink IP Address (if this information is available).
- Type **type**
  Audio sink type. Possible values are listed in the enumeration Type.
- std::string **subAddress**
- State **state**
  Indicates the current state of the sink.
- AnnouncementType **announcementType**
  Indicates the currnet announcement type that is playing on the sink.
- std::string **outputGain**
  Indicates the current output gain.
- bool **muteActive**
  Indicates if the sink is muted or not.
- int **monitoringSinkId**
  Indicates the sink Id that is currently monitored by this sink.
- HealthState **healthState**
  Indicates the health state of the sink.

9.5.3 Detailed Description

This class provides an interface to query the presence and status of visual devices in the system. Device attributes including device name, communications type and address, communications and health status can be queried.

9.5.4 Member Enumeration Documentation

9.5.4.1 enum VisualDisplay::CommsType

Method of communications to the display

ENUMERATION

▶**SERIAL** A serial interface is being used to communicate with the display device

▶**IP_V4** IP V4 is being used to communicate with the display device

- ▶**IP_V6** IP V4 is being used to communicate with the display device

9.5.4.2 enum VisualDisplay::HealthState

Health State of the display device.

ENUMERATION

- ▶**UNKNOWN** There is no communications to the display device and its health state cannot be determined.
- ▶**HEALTHY** There is communications to the display device and the display is reporting healthy. Displays that do not specifically report on the health status are also reported as healthy.
- ▶**FAULTY** There is communications to the display device and the display is reporting a fault.

9.5.4.3 enum VisualDisplay::HealthTestMode

Visual Display health test mode.

ENUMERATION

- ▶**TEST_DISABLED** Health tests disabled – no health tests are supported for the display
- ▶**TEST_ON_DEMAND** Health tests are enabled, and require manual or API initiation
- ▶**TEST_AUTOMATIC** Health tests are enabled, and conducted automatically (either continuous or periodic)

9.5.5 Member Function Documentation

9.5.5.1 string void getName() const

Queries the device name assigned to the display device.

RETURNS

- ▶***name*** device name assigned to the display device.

9.5.5.2 CommsType getCommsType() const

Retrieves the method used for communications with the device.

RETURNS

- ▶***CommsType*** *M*ethod used for communications with the device.

9.5.5.3 string getAddress() const

Returns the address used for communications to the display device.

RETURNS

▶ ***CommsType*** Address used for communications to the display device. Content of this is dependent on the communications type as described below:

SERIAL: Display address identifier, used to differentiate displays on serial buses

IP_V4: IP address (V4) of the display

IP_V6: IP address (V6) of the display

9.5.5.4 HealthState getHealthState() const

Returns health state of a VisualDisplay.

RETURNS

HealthState Health test mode of the specified VisualDisplay .

9.5.5.5 HealthTestMode getHealthTestMode() const

Returns health test mode for a VisualDisplay.

RETURNS

▶ ***HealthTestMode*** Health test mode of the specified VisualDisplay – valid values are:

- TEST_DISABLED: Tests are disabled
- TEST_ON_DEMAND: Tests only conducted on user request
- TEST_AUTOMATIC: Tests conducted automatically

9.5.5.6 void initiateHealthTest()

Initiates health test for the visual display.

The current state of health test can be observed using the getHealthTestStatus() method.

9.5.5.7 HealthTestStatus getHealthTestStatus() const

Returns the current health test state for the display.

RETURNS

▶ ***HealthTestStatus*** Health test status of the specified PaSink – valid values are:

- TEST_PENDING: Display is currently not Idle, and will be tested when next idle.
- TEST_NONE: Display is not being tested.
- TEST_IN_PROGRESS: Display testing in progress. Once the state changes from TEST_IN_PROGRESS to TEST_NONE, healthState of the unit can be queried.

9.5.5.8 void initiateHealthTest()

Initiates health test for the visual display.

The current state of health test can be observed using the getHealthTestStatus() method.

9.6 PaZone Class Reference

PaZone class represents a collection of PaSinks. These form system audio zones. Zones are part of the system configuration and can be customised by users to suit system audio zoning requirements.

Note that a zone can represent a fine a granularity such as a single speaker and as well as a larger area (eg. Factory floor, concourse, station, rail corridor). Zone ID and descriptions are non-volatile and will be consistent from one application invocation to the next as long as the configuration of the system has not changed.

9.6.1 Public Types

No types defined.

9.6.2 Public Attributes

- std::string **ID**
  ID of the PA zone. This is for the application to use as an Id to request an audio sink. For Example: “Sydney/North Shore Line/Chatswood/Platform 1”, “Vestibules”, “Upper Deck”.

9.6.3 Detailed Description

PaZone is a collection of audio sinks.

9.6.4 Constructor and Destructor Documentation

9.6.4.1 PaZone (const PaZone& zone)

Creates a duplicate copy of the PaZone object using the copy constructor.

PARAMETERS

- ▶**zone** another instance of the PaZone object which needs to be duplicated for internal storage.

9.6.5 Member Enumeration Documentation

No types defined.

9.6.6 Member Data Documentation

No types defined.

9.6.7 Member Function Documentation

9.6.7.1 PAController::PaSinkArray getMembers() const

Returns the list of member PaSinks.

RETURNS

- ▶***PAController::PaSinkArray*** List of PASink IDs.

9.6.7.2 void setMembers(const AudioServer::NumberArray& sinkIDList, bool isPersistent = false) throw (std::invalid_argument)

Assigns member PaSinks comprising a PaZone.

PARAMETERS

- ▶***sinkIDList*** List of PASink IDs.
- ▶***isPersistent*** If set to true, zone changes will be persistent across device restarts.

EXCEPTIONS

- ▶C++ std::invalid_argument
- ▶Java, IllegalArgumentException

If one of the specified PaSink IDs do not exist.

9.6.7.3 void mute()

Mute all members of the audio zone.

9.6.7.4 void unmute()

Unmutes all members of the audio zone.

9.6.7.5 bool isMuted() const

Queries audio output mute status of member audio sinks.

RETURNS

- ▶ ***bool*** True if all members are muted, false otherwise

9.7 PaTrigger Class Reference

PaTrigger is a software or hardware trigger that indicates the PaSource is now activated. When the trigger is active, PaSource will stream audio to connected PaSinks. Examples of triggers are on-screen Talk buttons, physical PTT buttons on microphones, digital inputs. This class provides an interface to retrieve information on a trigger including its priority.

9.7.1 Public Types

- enum **Type**
  - TRIGGER_TYPE_UNDEFINED = 0,
  - DIN_NORMALLY_OPEN,
  - DIN_NORMALLY_CLOSED,
  - KEYPAD_KEY,
  - VOX,
  - STREAM_PRESENCE,
  - TOUCHPAD_ZONE,
  - INCOMING_TELEPHONY_CALL,
  - MICROPHONE_CURRENT_SENSE
- enum **State**
  - UNKNOWN = 0,
  - INACTIVE,
  - ACTIVE

9.7.2 Public Attributes

- int **id**
  Id of the PA Trigger. This is for the application to use as an Id to query the trigger. For Example: 3.
- int **associatedSourceId**
  Id of the PA Source to which this trigger is bound to.
- std::string **label**
  Textual description (if available) of the trigger. For Example: "NAC DIO Port 1".
- Type **type**
  Trigger Type. Possible values are listed in the enumeration Type.
- std::string **subAddress**
- int **priority**
- State **state**
  Indicates the current state of the trigger.

9.7.3 Detailed Description

PaTrigger is used to define a trigger to enable/disable PA

9.7.4 Constructor and Destructor Documentation

9.7.4.1 PaTrigger (const PaTrigger &paTriggerAnother)

Creates a duplicate copy of the PaTrigger object using the copy constructor.

PARAMETERS

- **paTriggerAnother** another instance of the PaTrigger object which needs to be duplicated for internal storage

9.7.5 Member Enumeration Documentation

9.7.5.1 enum PaTrigger::Type

Trigger Type

ENUMERATION

- **TRIGGER_TYPE_UNDEFINED** Third party trigger
- **DIN_NORMALLY_OPEN** Digital Input which is normally open
- **DIN_NORMALLY_CLOSED** Digital Input which is normally closed
- **KEYPAD_KEY** Reserved for future implementation
- **VOX** Voice Activity Detected
- **STREAM_PRESENCE** Reserved for future implementation
- **TOUCHPAD_ZONE** Reserved for future implementation
- **INCOMING_TELEPHONY_CALL** Telephony call received
- **MICROPHONE_CURRENT_SENSE** Microphone detected

9.7.5.2 enum PaTrigger::State

Trigger State

ENUMERATION

- **UNKNOWN** Trigger state is unknown

- ▶**INACTIVE** Trigger state is currently inactive
- ▶**ACTIVE** Trigger state is currently active

9.7.6 Member Data Documentation

9.7.6.1 std::string PaTrigger::subAddress

Provides a device specific sub-address. For Example:

- ▶**Digital Input** Input Number
- ▶**Keypad Key** Key Number
- ▶**VOX** VOX Level
- ▶**Touch Pad Zone** Zone Number
- ▶**Incoming Telephony Call** On_Ring, On_DTMF_Number

9.7.6.2 int PaTrigger::priority

A value from 1-100 representing the priority with which PA announcement will be made when the trigger is activated. 1 is lowest priority and 100 is highest priority.

9.7.7 Member Function Documentation

PaTrigger has no member functions.

9.8 PaSelector Class Reference

PaSelector is a software or hardware zone selector switch. When activated, the audio source to zone routing is changed to reflect the selector position.

This class provides an interface to query information on a selector.

9.8.1 Public Types

- enum **Type**
  - SELECTOR_TYPE_UNDEFINED = 0,
  - DIN_NORMALLY_OPEN,
  - DIN_NORMALLY_CLOSED,
  - KEYPAD_KEY,
  - TOUCHPAD_REGION,
  - TELEPHONY_INPUT,
  - TELEPHONY_DID,
  - USER_DEFINED
- enum **Protection**
  - FIXED_CONFIGURATION = 0,
  - VOLATILE_SOFTWARE_CONFIGURABLE,
  - NONVOLATILE_SOFTWARE_CONFIGURABLE
- enum **Mode**
  - ADD_TO_EXISTING_SET = 0,
  - REPLACE_EXISTING_SET
- enum **DynamicRestriction**
  - UPDATES_NOT_ALLOWED = 0,
  - ADD_SINKS_TO_SET_ALLOWED,
  - ADD_REMOVE_SINKS_TO_SET_ALLOWED
- enum **State**
  - ENABLED_AND_UNKNOWN = 0,
  - ENABLED_AND_INACTIVE,
  - ENABLED_AND_ACTIVE,
  - DISABLED

9.8.2 Public Attributes

- int **id**
  Id of the PA Selector. Application can use this to enquire/modify the selector. For Example: 8.
- int **associatedSourceId**
  Id of the PA Source to which the selector is bound to.

- std::string **label**
  Textual description (if available) of the selector. For Example: "keypad Button 1".
- Type **type**
  Selector Type. Possible values are listed in the enumeration Type.
- std::string **number**
- Protection **protection**
  Selector Protection. Possible values are listed in the enumeration Protection.
- Mode **mode**
  Selector Mode. Possible values are listed in the enumeration Mode.
- DynamicRestriction **dynamicRestriction**
  Selector Dynamic Restriction. Possible values are listed in the enumeration DynamicRestriction.
- State **state**
  Indicates the current state of the selector.

9.8.3 Detailed Description

PaSelector is used to select pre-configured audio sinks.

9.8.4 Constructor and Destructor Documentation

9.8.4.1 PaSelector (const PaSelector &paSelectorAnother)

Creates a duplicate copy of the PaSelector object using the copy constructor.

PARAMETERS

▶ **paSelectorAnother** another instance of the PaSelector object which needs to be duplicated for internal storage

9.8.5 Member Enumeration Documentation

9.8.5.1 enum PaSelector::Type

Selector Type

ENUMERATION

▶ **SELECTOR_TYPE_UNDEFINED** Third party zone selector

▶ **DIN_NORMALLY_OPEN** Digital Input which is normally open

▶ **DIN_NORMALLY_CLOSED** Digital Input which is normally closed

▶ **KEYPAD_KEY** KEYPAD_KEY

- **TOUCHPAD_REGION** TOUCHPAD_REGION
- **TELEPHONY_INPUT** TELEPHONY_INPUT
- **TELEPHONY_DID** TELEPHONY_DID
- **USER_DEFINED** Software Selector

9.8.5.2 enum PaSelector::Protection

Selector Protection

ENUMERATION

- **FIXED_CONFIGURATION** Selector configuration cannot be altered/changed
- **VOLATILE_SOFTWARE_CONFIGURABLE** Selector configuration can be updated using software; however changes will not be saved
- **NONVOLATILE_SOFTWARE_CONFIGURABLE** Selector configuration can be updated using software and changes will be persistent

9.8.5.3 enum PaSelector::Mode

Selector Mode

ENUMERATION

- **ADD_TO_EXISTING_SET** Sink(s) can be added to existing set during announcement
- **REPLACE_EXISTING_SET** Sink(s) will replace existing set during announcement

9.8.5.4 enum PaSelector::DynamicRestriction

Selector DynamicRestriction

ENUMERATION

- **UPDATES_NOT_ALLOWED** Addition/Deletion of sinks not allowed
- **ADD_SINKS_TO_SET_ALLOWED** New sink(s) can be added to the selector set, however, sinks cannot be removed
- **ADD_REMOVE_SINKS_TO_SET_ALLOWED** Sink(s) can be added or removed from the selector set

9.8.5.5 enum PaSelector::State

Selector State

ENUMERATION

- **ENABLED_AND_UNKNOWN** Selector is enabled however it’s state is not known at time of query
- **ENABLED_AND_INACTIVE** Selector is enabled and is currently inactive
- **ENABLED_AND_ACTIVE** Selector is enabled and is currently active
- **DISABLED** Selector is disabled

9.8.6 Member Data Documentation

9.8.6.1 std::string PaSelector::number

Indicates device specific details for the selector. For Example:

- **Digital Input** Input Number
- **Keypad key** Key Number
- **Touch Pad Region** Region Number
- **Telephony Input** DTMF
- **Telephony DID** Indial Number

9.8.7 Member Function Documentation

PaSelector has no member functions.

9.9 PAControllerObserver Class Reference

The PAControllerObserver class provides an interface for client applications to be notified in the event of selected PA related events.

The client application needs to create a child class inheriting from PAControllerObserver, and override the methods the application is interested in.

Note the methods of the observer class will be executed in the context of a different thread, and the client applications needs to ensure all actions taken within the observer methods are thread-safe. In addition, all observer methods are executed within the same thread, observer methods are not intended to block. Recommended usage of the observer methods is to send a message to another thread in the client application to perform any long tasks.

9.9.1 Detailed Description

The PAControllerObserver provides an interface to observe changes in the PAController Class.

9.9.2 Public Types

- enum **MessageRetrievalError**
  - MRE_NO_ERROR = 0,
  - MRE_TIMED_OUT,
  - MRE_INVALID_PARAMS

9.9.3 Constructor and Destructor Documentation

9.9.3.1 virtual ~PAControllerObserver()

Public destructor.

9.9.4 Member Function Documentation

9.9.4.1 virtual void onPaSourceUpdate(PaSource source)

Callback function - called when a PA Source is updated. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

▶ ***source*** Reference to the PA Source that has changed state

9.9.4.2 virtual void onPaSourceDelete(int sourceId)

Callback function - called when a PA Source is deleted. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***sourceId*** Id of the PA Source removed from the list

9.9.4.3 virtual void onPaSinkUpdate(PaSink sink)

Callback function - called when a PA Sink is updated. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***sink*** Reference to the PA Sink that has changed state

9.9.4.4 virtual void onPaSinkDelete(int sinkId)

Callback function - called when a PA Sink is deleted. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***sinkId*** Id of the PA Sink removed from the list

9.9.4.5 virtual void onPaTriggerUpdate(PaTrigger trigger)

Callback function - called when a PA Trigger is updated. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***trigger*** Reference to the PA Trigger that has changed state

9.9.4.6 virtual void onPaTriggerDelete(int triggerId)

Callback function - called when a PA Trigger is deleted. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***triggerId*** Id of the PA Trigger removed from the list

9.9.4.7 virtual void onPaSelectorUpdate(PaSelector selector)

Callback function - called when a PA Selector is updated. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***selector*** Reference to the PA Selector that has changed state

9.9.4.8 virtual void onPaSelectorDelete(int selectorId)

Callback function - called when a PA Selector is deleted. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***selectorId*** Id of the PA Selector removed from the list

9.9.4.9 virtual void onAudioMessageRetrievalComplete(const std::string& requestID, bool success, PAControllerObserver::MessageRetrievalError errorCode, const std::string& uri)

Callback function - called retrieval of audio message due to a previous call to Message::retrieveAudioMessage is complete.

Note this method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***requestID*** Identifier assigned to the audio content retrieval request by the previous invocation to Message::retrieveAudioMessage
- ***success*** Indicates if audio message could be retrieved. True indicates success and false indicates failure.
- ***errorCode*** Set to one of the supported enumerated error codes on failure.
- ***uri*** HTTP URI that can be used to access the audio message contents.

10 NetSpire SDK Class Reference - Call Management

10.1 Overview

This section provides information on API models provided for monitoring and controlling the call management subsystem.

The main classes provided in the model are:

CallController: Provides an interface to query current calls, initiate new calls, and manage (e.g. hold/transfer/terminate) existing calls.

CallInfo: Provides information on an individual call.

TerminalInfo: An entity that has been involved in a call. These can include Intercom devices, desk or wall mounted telephones, SIP endpoints, or parties contacted through an ISDN or SIP trunk.

ISDNTrunk/SIPTrunk: Connections to external phone systems.

10.2 CallController Class Reference

Provides an interface to query current calls, initiate new calls, and manage (e.g. hold/transfer/terminate) existing calls.

10.2.1 Public Types

- typedef std::vector< CallInfo > **CallInfoArray**
- typedef std::vector< CDRInfo > **CDRInfoArray**
- typedef std::vector< TerminalInfo > **TerminalInfoArray**
- typedef std::vector< std::string > **StringArray**

10.2.2 Detailed Description

An instance of this class can be used to control call flow and handle incoming and outgoing calls in a system.

10.2.3 Member Typedef Documentation

10.2.3.1 typedef std::vector<CallInfo> CallController::CallInfoArray

A collection of CallInfo objects.

10.2.3.2 typedef std::vector<CDRInfo> CallController::CDRInfoArray

A collection of CDRInfo objects.

10.2.3.3 typedef std::vector<TerminalInfo> CallController::TerminalInfoArray

A collection of Terminal objects.

10.2.3.4 typedef std::vector<std::string> CallController::StringArray

A collection of strings. Note: The "clear" method is required to be called when the StringArray is declared with size specified, for example, new StringArray(8). The "clear" method needs to be called once, before any "add" actions.

10.2.3.5 typedef std::vector<SIPTrunk> CallController::SIPTrunkList

A collection of SIP Trunk objects.

10.2.3.6 typedef std::vector<ISDNTrunk> CallController::ISDNTrunkList

A collection of ISDN Trunk objects.

10.2.4 Member Function Documentation

10.2.4.1 void CallController::createDestination(const StringArray &bPartyExtList, const std::string &CLI, const std::string &deviceID) throw (std::invalid_argument)

Creates a destination map on the CXS (Communication Exchange Server). A destination map is a set of rules to handle incoming calls on the server. When a call is originated from a caller, or a group of callers, the rules decide which destination to route the calls to.

PARAMETERS

- ***bPartyExtList*** List of Group Numbers to redirect incoming calls. The list should always contain at least one entry. The system will initiate a call to the first entry in the list. If the call is not answered within a predefined timeout period it will be escalated to the next entry in the list.

- ***CLI*** Caller Line Identification of the incoming call as received by one of the call control servers (normally CXS1 and CXS2). If dynamic routing is enabled for a phone (e.g. as determined by an associated discrete signal), the CLI will be the phone ID followed by a suffix. The suffix will be set to 0 if the Digital Input signal associated with the phone is not asserted or is unknown. The suffix will be set to 1 if the Digital Input signal associated with the phone is asserted.

- ***deviceID*** Device ID of the CXS where the call will be received. The only valid device type that can be specified for this field is CXS.

EXCEPTIONS

- ***std::invalid_argument*** This exception is thrown when an invalid ID number or device id is specified.

10.2.4.2 int CallController::createCall(const std::string &aPartyCLI, const std::string &bPartyCLI) throw (std::invalid_argument)

A new call is created between A-Party to B-Party. A call is first made to A-Party, and on answering, it is bridged to B-Party – once B-party answers, call is established between the two answerCallOnTerminal.

PARAMETERS

- ***aPartyCLI*** Caller Line Identification of A-Party
- ***bPartyCLI*** Caller Line Identification of B-Party

RETURNS

- Call reference ID which uniquely identifies this call

EXCEPTIONS

- ***std::invalid_argument*** This exception is thrown when an invalid CLI is specified.

10.2.4.3 void CallController::answerCallOnTerminal(int callReference, const std::string &bPartyCLI, const std::string &aPartyCLI) throw (std::invalid_argument)

An incoming call is answered on the specified terminal. User MUST specify at least one of these two optional arguments: callReference and aPartyCLI

PARAMETERS

- ***callReference*** unique identifier of a call. Is conditonally optional and can be set to 0 if not present.
- ***bPartyCLI*** Caller Line Identification of B-Party. Is mandatory.
- ***aPartyCLI*** Caller Line Identification of A-Party Is conditionally optional and can be left as empty if not present.

RETURNS

- Call reference ID which uniquely identifies this call

EXCEPTIONS

- **std::invalid_argument** This exception is thrown when an invalid CLI is specified.

10.2.4.4 void CallController::resumeCall(int callReference, const std::string &bPartyCLI) throw (std::invalid_argument)

A held call is resumed. This terminal needs to have previously put the call on hold. Once executed, the terminal will be connected back to the caller.

PARAMETERS

- ***callReference*** call ID as specified in CallInfo. Call ID is optional and can be set to 0.
- ***bPartyCLI*** B-party Caller Line Identification to be resumed. B-party Caller Line Identification is mandatory.

EXCEPTIONS

- **std::invalid_argument** This exception is thrown when an invalid call reference or CLI is specified.

10.2.4.5 void CallController::resumeCallOnTerminal(int callReference, const std::string &bPartyCLI, const std::string &aPartyCLI) throw (std::invalid_argument)

A held call is resumed on the specified terminal. This terminal may or may not have previously put the call on hold. Once executed, the terminal will be connected to the caller.

PARAMETERS

- ***callReference*** unique identifier of a call. Is conditonally optional and can be set to 0 if not present.
- ***bPartyCLI*** Caller Line Identification of B-Party. Is mandatory.
- ***aPartyCLI*** Caller Line Identification of A-Party Is conditionally optional and can be left as empty if not present.

RETURNS

- Call reference ID which uniquely identifies this call

EXCEPTION

- **std::invalid_argument** This exception is thrown when an invalid CLI is specified.

10.2.4.6 void CallController::transferCall(int callReference, const std::string &bPartyCLI) throw (std::invalid_argument)

A-party leg of an active call is transferred to the specified new B-Party. Note call transfer support is dependent on system architecture.

PARAMETERS

- ***callReference*** call ID as specified in CallInfo
- ***bPartyCLI*** B-party Caller Line Identification where the call will be redirected

EXCEPTIONS

- ***std::invalid_argument*** This exception is thrown when an invalid call reference or CLI is specified.

10.2.4.7 void CallController::holdCall(int callReference, const std::string &bPartyCLI) throw (std::invalid_argument)

A-party leg of an active call is put on hold. B-party is then able to accept other incoming calls.

PARAMETERS

- ***callReference*** call ID as specified in CallInfo. Call ID is optional and can be set to 0.
- ***bPartyCLI*** B-party Caller Line Identification to be put on hold. B-party address is mandatory.

EXCEPTIONS

- ***std::invalid_argument*** This exception is thrown when an invalid call reference or CLI is specified.

10.2.4.8 void CallController::terminateCall(int callReference, const std::string &callConnection) throw (std::invalid_argument)

Terminates the call identified by the call reference.

PARAMETERS

- ***callReference*** call ID as specified in CallInfo
- ***callConnection*** send empty string to release all connections. To release a specific connection (e.g. current B-party) but leave the call active in the system specify the address of the connection to be released. Note support for releasing a specific connection is dependent on system architecture.

EXCEPTIONS

- ***std::invalid_argument*** This exception is thrown when an invalid call reference or call leg is specified.

10.2.4.9 void CallController::terminateCall(int callReference, int cleardownCauseCode) throw (std::invalid_argument)

A ringing/connected call is terminated with the ability to provide a cleardown cause code. The specification which defines the values and meanings of the Cleardown Cause Codes is the ITU (International Telecommunication Union) specification Q.850.

PARAMETERS

- ***callReference*** call ID as specified in CallInfo
- ***cleardownCauseCode*** code to indicate the reason why the call cleared down.

EXCEPTIONS

- ***std::invalid_argument*** This exception is thrown when an invalid call reference is specified.

10.2.4.10 CallController::CallInfoArray CallController::getCalls()

List all active calls in the system.

RETURNS

- Current active calls.

10.2.4.11 CallController::CDRInfoArray CallController::getCDRMessages()

List all active call detail records in the system.

RETURNS

- Current call logs

10.2.4.12 CallController::TerminalInfoArray CallController::getTerminalList()

List all terminals in the system.

RETURNS

- Current active calls

10.2.4.13 void CallController::registerObserver(CallControllerObserver* callObserver)

Registers an observer with the CallControllerClass instance. Callback methods on the observer will be called on state changes.

PARAMETERS

- ***callObserver*** Observer instance

10.2.4.14 CallController::SIPTrunkList CallController::getSIPTrunks()

Returns a list of SIP Trunks that are configured in the system using CXS server web-interface.

PARAMETERS

- None

10.2.4.15 CallController::ISDNTrunkList CallController::getISDNTrunks()

Returns a list of ISDN Trunks that are configured in the system using CXS server web-interface.

PARAMETERS

- None

10.3 CallInfo Class Reference

The CallInfo class provides an interface to query information on an individual call. Information includes parties involved in the call and current call state.

10.3.1 Public Types

- enum **State**
  - PROGRESS = 0,
  - CONNECTED,
  - HELD,
  - DISCONNECTED,
  - UNKNOWN

10.3.2 Public Attributes

- unsigned long **called**
  Unique ID for the call
- std::string **callStartTime**
  Call origination time. Format of the string is YYYY-MM-DD HH:MM:ss
- unsigned long **callDuration**
  Call duration
- unsigned long **callAnswerTime**
  Time elapsed before call is accepted by B-party
- std::string **callAPartyId**
  A-party (call originator) address
- std::string **callBPartyId**
  B-party (recipient) address
- std::string **callTargetIds**
  List of addresses currently being alerted in relation to this call
- State **callState**
  Call State
- std::string **callStateText**
  Call State Text

  int **callType**
  Call Type - Refer to Table 19
- std::string **callTypeText**

  Call Type Text - Refer to Table 19

- int **callReleaseCause**

Different supported call types, values and corresponding description follows:

Call Type	Value	Description
PEI_TYPE1	0	• Type 1 call initiated by a Passenger Emergency Intercom(PEI) or Help Point (HP) • Call types can be classified by end users to differentiate different call types e.g. Informational or Emergency
ICOM	1	• Call initiated by an operator, such as, IP based public address control unit (IPPA), Crew Panel (CP), Crew Intercom(CI), Generic SIP device
TR	3	• Train Radio/IP call initiated by Wayside to crew (via a CP/CI) on the train
TR_PA	4	• Train Radio Public announcement call on the train initiated by wayside to the train
TR_EMERGENCY	5	• Train Radio Emergency call – from Wayside to crew
PEI_TYPE2	7	• Type 2 call initiated by a Passenger Emergency Intercom(PEI) or Help Point (HP) • Call types can be classified by end users to differentiate different call types e.g. Informational or Emergency
CP2TR	9	• Train Radio/IP call initiated by crew (via a CP/CI) on the train to Wayside
UNKNOWN	10	• An unknown call type

Table 54 - Call Types

10.3.3 Detailed Description

CallInfo objects are used to provide information on active calls managed by the system.

10.3.4 Member Enumeration Documentation

10.3.4.1 enum CallInfo::State

Defines call states reported by the system.

ENUMERATION

▶**PROGRESS** The call is ringing (may also be known as alerting)

▶**CONNECTED** The call is connected

▶**HELD** The call is on hold

▶**DISCONNECTED** The call is disconnected

▶**UNKNOWN** The system is unable to determin the call state

10.3.5 Constructor and Destructor Documentation

10.3.5.1 CallInfo(const CallInfo &callInfoAnother)

Creates a duplicate copy of the CallInfo object using the copy constructor.

PARAMETERS

- **callInfoAnother** another instance of the CallInfo object which needs to be duplicated for internal storage

10.3.6 Member Data Documentation

10.3.6.1 int CallInfo::callReleaseCause

Call Release Cause as specified in ITU-T Q.850. Refer to link http://networking.ringofsaturn.com/Routers/isdncausecodes.php for a full list of release causes. Common release causes are:

- **16** Normal call clearing
- **17** User busy
- **18** No user responding
- **19** No answer from user (user alerted)
- **21** Call Rejected
- **31** Normal, unspecified

10.4 CDRInfo Class Reference

Provides and interface to query Call Detail Record information on a past call. Information includes parties involved in the call, duration of call and hangup cause.

10.4.1 Public Attributes

- int **id**
  Represents the unique id of the CDR record
- int **startDateYear**
  Indicates the year when the CDR record was created
- int **startDateMonth**
  Indicates the month when the CDR was created
- int **startDateDay**
  Indicates the day when the CDR was created
- int **startTimeHour**
  Indicates the hour when the CDR was created
- int **startTimeMinute**
  Indicates the minute when the CDR was created
- int **startTimeSecond**
  Indicates the second when the CDR was created
- int **callDuration**
  Indicates the duration of the call in seconds
- int **answerDuration**
  Indicates the duration of the answer time in seconds
- std::string **callFromTermId**
  Indicates the terminal id from where the call was placed
- std::string **callToTermIdList**
  Indicates a list of one or more terminal ID where the call is targeted. Terminal IDs are comma seperated
- int **callCompletionStatus**
  Indicates the completion code for the call.

10.4.2 Detailed Description

Call Detail Record (CDR) is used to provide information on call logs managed by the Communications Exchange Server. Information such as call start date and time, caller id, callee id, call duration and call hangup cause can be retrieved from the system. The Communications Exchange Server will maintin the 100 call records after which the oldest call record will be removed to insert new call records.

10.4.3 Constructor and Destructor Documentation

10.4.3.1 CDRInfo (const CDRInfo &cdrinfoAnother)

Creates a duplicate copy of the CDRInfo object using the copy constructor.

PARAMETERS

cdrinfoAnother another instance of the CDRInfo object which needs to be duplicated for internal storage.

10.5 TerminalInfo Class Reference

Provides information on an entity that is involved in a call. These can include Intercom devices, desk or wall mounted telephones, SIP endpoints, or parties contacted through an ISDN or SIP trunk.

10.5.1 Public Attributes

- int **termId**
  Represents the unique id of the terminal
- std::string **termInfo**
- std::string **termAddress**
  Indicates the IP address of the terminal
- std::string **termType** – Refer to Table 20
- std::string **termLocation**
- int **termState**

  ENUMERATION

  - **UNKNOWN** The call state is not known by the system
  - **FAULTY** The terminal has a known fault
  - **IDLE** The terminal is not involved in a call
  - **RINGING** The terminal is ringing
  - **CONNECTED** The terminal is connected to a call
  - **HELD** The terminal has a call on hold
  - **ISOLATED** The terminal has been isolated from the sytem
- std::string **termStateText**
- int **termConnectedToPartyId**
- int **termOnHoldByPartyId**

10.5.2 Detailed Description

Different supported terminal types, values and corresponding description follows:

Terminal Type Name	Description
Operator	• Netspire Crew Intercom / Crew Panel
Help Point/Intercom	• Netspire Help Point device • Netspire Passenger Emergency Intercom
IPPA	• Netspire IP based Public Address control unit
3rd Party Ext	• Generic Help Points • Generic SIP devices
Unknown	• Unknown device

Table 55 - Terminal Types

10.5.3 Constructor and Destructor Documentation

10.5.3.1 TerminalInfo (const TerminalInfo &termInfoAnother)

Creates a duplicate copy of the TerminalInfo object using the copy constructor.

PARAMETERS

▶**termInfoAnother** another instance of the TerminalInfo object which needs to be duplicated for internal storage

10.6 SIPTrunk Class Reference

Provides information including user assigned name and current status of a SIP connection to an external phone system.

10.6.1 Public Attributes

None

10.6.2 Detailed Description

The SIPTrunk provides information about the SIP interface that is configured in the CX server.

10.6.3 Member Enumeration Documentation

10.6.3.1 enum SIPTrunk::Type

Trunk type enumerates the list of different trunk technologies that can be configured using the web based administration interface.

ENUMERATION

- **SIP** SIP Trunk – a telephony trunk to an external SIP (Session Initiation Protocol) server
- **E1T1** Primary rate E1 or T1 ISDN interface

10.6.4 Constructor and Destructor Documentation

10.6.4.1 SIPTrunk (const SIPTrunk &anotherSIPTrunk)

Creates a duplicate copy of the SIPTrunk object using the copy constructor.

PARAMETERS

- **anotherSIPTrunk** another instance of the SIPTrunk object which needs to be duplicated for internal storage

10.6.5 Member Function Documentation

10.6.5.1 virtual std::string SIPTrunk::getDeviceID() const

Returns the device where the trunk is installed and configured. This is typically a CX server.

PARAMETERS

- None

10.6.5.2 virtual SIPTrunk::Type SIPTrunk::getType()

Returns the type of trunk. In this case the type is SIP and the value returned is 0.

PARAMETERS

- None

10.6.5.3 virtual std::string SIPTrunk:: getType () const

Returns the name assigned to the trunk in the web based administration interface.

PARAMETERS

- None

10.6.5.4 virtual int SIPTrunk::getStatus() const

Returns the current status of the SIP Trunk. The return values are:

1 – SIP Registration is successful.

2 – Not Registered

3 – SIP Registration failed.

PARAMETERS

- None

10.7 ISDNTrunk Class Reference

Provides information including user assigned name and current status of a E1 connection to an external phone system.

10.7.1 Public Attributes

None

10.7.2 Detailed Description

The ISDNTrunk provides information about the ISDN interface that is configured in the CX server.

10.7.3 Member Enumeration Documentation

10.7.3.1 enum ISDNTrunk::Type

Trunk type enumerates the list of different trunk technologies that can be configured using the web based administration interface.

ENUMERATION

- ▶**SIP** SIP Trunk – a telephony trunk to an external SIP (Session Initiation Protocol) server
- ▶**E1T1** Primary rate E1 or T1 ISDN interface

10.7.4 Constructor and Destructor Documentation

10.7.4.1 ISDNTrunk (const ISDNTrunk &anotherISDNTrunk)

Creates a duplicate copy of the ISDNTrunk object using the copy constructor.

PARAMETERS

- ▶**anotherISDNTrunk** another instance of the ISDNTrunk object which needs to be duplicated for internal storage

10.7.5 Member Function Documentation

10.7.5.1 virtual std::string ISDNTrunk::getDeviceID() const

Returns the device where the trunk is installed and configured. This is typically a CX server.

PARAMETERS

▶None

10.7.5.2 virtual ISDNTrunk::Type ISDNTrunk::getType()

Returns the type of trunk. In this case the type is E1/T1 and the value returned is 1.

PARAMETERS

▶None

10.7.5.3 virtual std::string ISDNTrunk::getName() const

Returns the name assigned to the trunk in the web based administration interface.

PARAMETERS

▶None

10.7.5.4 virtual int ISDNTrunk::getLayer1Status() const

Returns the current status of the E1 Layer 1. The return values are:

1 – Link is up.

2 – Link is down.

PARAMETERS

▶None

10.7.5.5 virtual int ISDNTrunk::getLayer2_3Status() const

Returns the current status of the E1 Layer 2 and 3. The return values are:

1 – Link is up.

2 – Link is down.

PARAMETERS

▶None

10.8 CallControllerObserver Class Reference

The CallControllerObserver class provides an interface for client applications to be notified in the event of selected call related events.

The client application needs to create a child class inheriting from CallControllerObserver, and override the methods the application is interested in.

Note the methods of the observer class will be executed in the context of a different thread, and the client applications needs to ensure all actions taken within the observer methods are thread-safe. In addition, all observer methods are executed within the same thread, observer methods are not intended to block. Recommended usage of the observer methods is to send a message to another thread in the client application to perform any long tasks.

10.8.1 Detailed Description

The CallControllerObserver provides an interface to observe changes in the call controller class.

10.8.2 Constructor and Destructor Documentation

10.8.2.1 virtual CallControllerObserver::~CallControllerObserver()

Callback function - called when deleting the observer object

10.8.3 Member Function Documentation

10.8.3.1 virtual void AudioServerObserver::onCallUpdate(CallInfo callInfo)

Callback function - called when a new call is created or existing call parameters updated. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***callInfo*** to the new call added or existing call parameters updated

10.8.3.2 virtual void AudioServerObserver::onCallDelete(int callId)

Callback function - called when a call is deleted from the system. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

- ***callId*** of the call deleted from the system

10.8.3.3 virtual void AudioServerObserver::onCDRMessageUpdate(CDRInfo cdrInfo)

Callback function - called when a new Call Detail Record (CDR) is created or existing call parameters updated. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

10.8.3.4 *cdrInfo* of the Call Detail Record been updatedvirtual void AudioServerObserver::onCDRMessageDelete(int cdrId)

Callback function - called when a CDR is deleted from the system. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

cdrId id of the Call Detail Record been deleted

10.8.3.5 virtual void AudioServerObserver::onTerminalUpdate(TerminalInfo termInfo)

Callback function - called when a new Terminal is created or its parameters updated. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

TerminalInfo terminal information of the terminal been updated

10.8.3.6 virtual void AudioServerObserver::onTerminalDelete(int termId)

Callback function - called when a Terminal is deleted from the system. This method will be executed in a separate thread created by the SDK library.

PARAMETERS

termId id of the terminal been deleted

11 NetSpire SDK Class Reference – Passenger Information Server

11.1 Overview

The NetSpire SDK provides an interface for managing passenger information, incorporating support for vehicle positioning, track signalling and other signalling interfaces and systems.

The signalling information is supported by an abstracted interface to support a range of signalling systems. Support for a singallng system can either be via a standard NetSpire signalling driver or via an externally implemented signalling driver. The standard signalling drivers and interfaces supported in NetSpire is being continually updated, please contact Open Accesss if information relating to specific signalling system is required. The abstracted interface can also be used for third party development of signalling drivers and interfaces.

11.2 Working with a standard driver

To utilise a standard NetSpire signalling dirver, it must be first installed and configured specifically for the environment. The installation and configuration method varies by driver type and is not covered in this document.

When using a standard signalling driver, an external application can access the NetSpire passenger information subsystem through an instance of the *PassengerInformationServer* class. Accesses to the *PassengerInformationServer* instance is by using the method *AudioServer::getPassengerInformationServer().* The object acts as the master container for passenger information as it is implemented in the signalling driver. The signalling driver is able to support data for describing vehicles, stations, stops, platforms, trips and services. The actual data provided the signalling driver varies by signalling type, environment and implementation. This needs to be determined prior to using this interface.

The classes *PassengerInformationServe*r and *PassengerInformationObserve*r provide the means for interacting, controlling and monitoring the passenger information subsystem.

Class	Description
Passenger InformationServer	Class supporting all objects and methods associated with the provision of passenger information in the system.
PassengerInformationObserver	Provides an event based notification interface for passenger information related changes

In most environments the “*Vehicle*”, “*Station*” and “*Trip*” objects are used for maintaining key information relating to passenger information.

Object	Description

Vehicle	When used, represents a vehicle that moves from a source to a destination.
Station	Represents a location at which a vehicle stops during a Trip (journey).
Trip	An ordered pattern of defined location that a vehicle travels past during a Trip from a source to a destination. A Trip includes all locations where the vehicles passes regardless of whether it stops.

The following table provides a list of data types used by the API.

Object	Description
PlatformInfo	Represents a location within a station where passengers board the vehicle. Information contained with this class may include platform number, name and other detail relating to where the vehicle stops in a station / stop.
Service	A specific instance of a Trip with a start time. A daily operating timetable will comprise multiple Services.Once Trip may also apply to multiple Services.
TripStop	A location at which a vehicle stops during a Trip. This data type may include information regarding the amount of time it takes for vehicle to reach a stop. Where a time is specified, it is relative rather than absolute.
ServiceStop	Extension of TripStop as it applies to a Service. This includes absolute timing information about when the Vehicle will arrive at a stop. The ServiceStop supports dynamic modiciation to allow for delays and variation in actual train arrival and departure
Line	A descriptive non-functional label that can be associated with a list of stations / locations typically used in a Trip. The Lines datatype may be used for display and clarification purposes.
Priority	Classifies the type of Service by the locations it stops at during a trip. Examples include All Stops, Limited Stops andExpress services.

NetSpire supports a range of different signalling and timetabling systems and may be used in a variety of different transport environments. The classes and objects should be mapped contextually in accordance with the envrionments. For example, in a rail environment "***Vehicle***" objects may be used to represent trains and the ***"Station***" class are used for stations. In Water transportation, ***Vehicles*** may represent ferries and S*tations* may be associated with a wharf. For Bus transport systems, a ***Vehicle*** *may* represent a bus and the ***Station*** may be used for a bus stop. The rest of this document will refer to the system as a rail-based system for simplicity of descriptions. Therefore a "***Trip***" is defined as a pattern of defined stations that a vehicle travels from one source to a destination.

Stations may be associated with one or more platforms and in specific instances, sub platforms(bays). A **Service** is a specific instance of a trip that runs once at a given date and time. Service also maintains additional information specific to the service operation as defined in the class documentation below.

Differentiation between Trip and Service must be emphasised. Trips are stopping patterns of vehicles; these are defined by the transport authority as part of the scheduled timetable and are changed infrequently (i.e. standard working timetable "SWTT"). On the other hand, Services could easily be changed due to short term and daily operational requirements (ie daily working timetable "DWTT").

A ***Trip*** consists of a sequence of ***TripStop'***s where the vehicle passes during its journey. It also includes the **Line** assocated with the trip. The stopping pattern classifies the type of service (eg Express) and is designated as "**Priority**". in the system. Line and Priority are not applicable to all systems, for example systems without timetabling (Metro and monorail) may not utilise these structures.

A TripStop includes information for each stop in a trip. This information includes the station, the platform within that station.

For each stop in a trip, there is an expected arrival and departure time that is stored as an absolute offset from trip commencement time. By default, the dwell time may be calculated for each stop as the difference between scheduled arrival and departure offsets. All timing offsets for a trip are measured in seconds from trip commencement.

The Service object is an instantiation of a Trip and adds additional information including the state of the service and the vehicle ID providing the service. The vehicle ID is associated with the physical asset and the information is typically retrieved from the signalling system. The Service object is updated as the vehicle progresses through stops. The Service object also maintains details of the last station the Service has stopped at or passed. At Service commencement, this will be Service departing station. At Service termination, this will be the terminating station.

Details of stops during a Service are maintained in the ServiceStops object. By default, the ServiceStops includes information relating to estimated arrival time and departure times as per the predefined schedule of arrival/departure offsets in the Service. In the event it is needed to revise these during Service operation, an external application can dynamically modify this information using any external sensors (e.g. GPS, or external connectivity to a signalling system) that it may have available.

When invoked, the "setter" functions are internally validated by the system and may not be impacted if the changes will result in unsupported scenarious or if the user invoking the changes does not have sufficient privileges to modify the data.

The event driven call back mode of operation is the preferred method of receiving asynchronous update events about changes in the system. Asynchonous notification of events are accessed by subclassing the *PassengerInformationObserve*r class.

Alternatively, after performing a change using a “setter” function, in order to verify that a requested change has been accepted, the application should retrieve and validate the information via the associated “getter” functions. Note that a delay should be used before validating requested updates using the getter function. The magnitude of the delay will vary in accordance with the network size, topology and activity level. Typically all data would be updated within 500ms, however the appropritate delays should be validated during system trial. It is preferable to use the “getter” functions after receiving an asynchronous event notification for the data being updated, instead of using specific delay values. Event notifications are only be raised once the update is processed by the backend servers and the API connection is updated with the new information.

11.3 PassengerInformationServer Class Reference

The interface to the Passenger Information Server is encapsulated by the PassengerInformationServer class. The instance of the PassengerInformationServer class is obtained using the AudioServer::getPassengerInformationServer() method.

11.3.1 Member Typedef Documentation

11.3.1.1 typedef std::string LocationId

LocationId defines a location within the network. The location is specific to the network and system, and is used to store station IDs/names, bus stop IDs, track circuits, as well as pre-defined GPS area if applicable.

11.3.2 Member Enumeration Documentation

11.3.2.1 enum LocationRetrievalMethod

Specifies the method used for retrieving location and positioning information.

Enumeration

- **SIGNALLING** – **Signalling** - Location and positioning information is received using a Signalling system interface in the NetSpire system. Only the "getter" API methods of the PassengerInformationServer are enabled.
- **API_TC** - **Track Circuits**: Location and positioning information is received from the API. LocationId contains Track Circuit IDs..
- **API_PLAT – Platform**: Location and positioning information is received from the API. LocationId contains Station and Platform IDs..
- **API_GPS - GPS -** Location and positioning information is received from the API. LocationId contains GPS coordinates.

11.3.3 Member Function Documentation

11.3.3.1 bool PassengerInformationServer::getServerStatus()

Returns the current status of the signalling server (online/offline). Only applicable when the LocationRetrievalMethod is set to SIGNALLING.

RETURNS

Returns current status of the signalling server.

11.3.3.2 void PassengerInformationServer::setLocationRetrievalMethod (LocationRetrievalMethod method)

Sets the current location retrieval method to one of the supported methods. The use of the SIGNALLING method requires the installation of an interface application module for communicating with a signalling system.

PARAMETERS

method The location retrieval method to use.

11.3.3.3 LocationRetrievalMethod PassengerInformationServer::getLocationRetrievalMethod ()

Returns the current location retrieval method.

RETURNS

Returns an enumerated value containingthe current location retrieval method.

11.3.3.4 bool PassengerInformationServer::isAutomaticMessageGenerationEnabled ()

Returns the current status of the automatic message generation system.

RETURNS

Returns true if automatic message generation system is enabled; false if disabled.

11.3.3.5 void PassengerInformationServer::enableAutomaticMessageGeneration ()

Enables the automatic message generation system. When enabled, the PassengerInformationController in the NetSpire CXS servers will generate audio and visual messages based on available service information.

11.3.3.6 void PassengerInformationServer::disableAutomaticMessageGeneration ()

Disables the automatic message generation system. The PassengerInformationController in the NetSpire CXS servers will no longer generate audio and visual messages based on service information. This method can be used if the controller API application is required to generate specificialized or customised messages.

11.3.3.7 void PassengerInformationServer::setLines(const std::list<Line>& lines)

Replaces the Lines registered in the system with a new set of Lines.

PARAMETERS

lines Collection of lines to set

11.3.3.8 void PassengerInformationServer::addLine(const Line& line)

Registers a new Line in the system.

PARAMETERS

line New Line to register.

11.3.3.9 std::vector<Line> PassengerInformationServer::getLines()

Returns the list of all the lines defined in the system

RETURNS

Returns the list of all the lines defined in the system

11.3.3.10 void PassengerInformationServer::setServices(const std::list<Service>& services)

Replaces the Services registered in the system with a new set of Services.

PARAMETERS

services Collection of Services to set

11.3.3.11 void PassengerInformationServer::addService(const Service& service)

Registers a new Service in the system.

PARAMETERS

service New Service to register.

11.3.3.12 void PassengerInformationServer::deleteService(int serviceId)

Deletes a registered Service from the system. Any ServiceStops that are associated with the Service will also be deleted automatically.

PARAMETERS

serviceId Service to delete.

11.3.3.13 std::vector<Service> PassengerInformationServer::getServices()

Returns the list of all the services defined in the system

RETURNS

Returns the list of all the services defined in the system

11.3.3.14 void PassengerInformationServer::setServiceStops(const std::list<ServiceStop>& serviceStops)

Replaces the ServiceStops registered in the system with a new set of ServiceStops.

PARAMETERS

serviceStops Collection of ServiceStops to set

11.3.3.15 void PassengerInformationServer::addServiceStop(const ServiceStop& serviceStop)

Registers a new ServiceStop in the system.

PARAMETERS

serviceStop New ServiceStop to register.

11.3.3.16 void PassengerInformationServer::deleteServiceStop(int serviceStopId)

Deletes a registered ServiceStop from the system.

PARAMETERS

serviceStopId ServiceStop to delete.

11.3.3.17 std::vector<ServiceStop> PassengerInformationServer::getServiceStops()

Returns the list of all the service stops defined in the system

RETURNS

Returns the list of all the service stops defined in the system

11.3.3.18 void PassengerInformationServer::setVehicles(const std::list<Vehicle>& vehicles)

Replaces the Vehicles registered in the system with a new set of Vehicles.

PARAMETERS

vehicles Collection of Vehicles to set

11.3.3.19 void PassengerInformationServer::addVehicle(const Vehicle& vehicle)

Registers a new Vehicle in the system.

PARAMETERS

vehicle New Vehicle to register.

11.3.3.20 std::vector<Vehicle> PassengerInformationServer::getVehicles()

Returns the list of all the vehicles defined in the system

RETURNS

Returns the list of all the vehicles defined in the system

11.3.3.21 void PassengerInformationServer::setStations(const std::list<Stations>& stations)

Replaces the Stations registered in the system with a new set of Stations.

PARAMETERS

stations Collection of Stations to set

11.3.3.22 void PassengerInformationServer::addStation(const Station& station)

Registers a new Station in the system.

PARAMETERS

station New Station to register.

11.3.3.23 std::vector<Station> PassengerInformationServer::getStations()

Returns the list of all the stations defined in the system

RETURNS

Returns the list of all the stations defined in the system

11.3.3.24 std::vector<PlatformInfo> PassengerInformationServer::getPlatforms()

Returns the list of all the platforms defined in the system

RETURNS

Returns the list of all the platforms defined in the system

11.3.3.25 std::vector<Trip> PassengerInformationServer::getTrips()

Returns the list of all trips defined in the system.

RETURNS

Returns the list of all trips defined in the system

11.3.3.26 std::vector<TripStop> PassengerInformationServer::getTripStops()

Returns the list of all trip stops defined in the system.

RETURNS

Returns the list of all trip stops defined in the system

11.3.3.27 void PassengerInformationServer::registerObserver(PassengerInformationObserver* observer)

Registers an observer with the PassengerInformationServer instance. Callback methods on the observer will be called to notify when information and state changes in the system.

PARAMETERS

observer Observer instance

11.4 PassengerInformationObserver Class Reference

The PassengerInformationObserver provides an interface to observe changes in the passenger information system including object status changes.

The application must create a subclass of the PassengerInformationObserver and register an object of the new class using the PassengerInformationServer::registerObserver method.

The notification methods on the observer class are then invoked by the API, which can be used to keep track of status updates.

11.4.1 Member Function Documentation

11.4.1.1 virtual void PassengerInformationObserver::onServerStatusUpdated(bool status)

This function is called back when the status of the signalling server is updated.

PARAMETERS

- ▶ **status** signalling server status

11.4.1.2 virtual void PassengerInformationObserver::onLineUpdated(Line line)

This function is called back when a new line is defined or an existing line is updated in the system.

PARAMETERS

- ▶ **line** Line object.

11.4.1.3 virtual void PassengerInformationObserver::onLineRemoved(Line line)

This function is called back when an existing line is removed from the system.

PARAMETERS

- ▶ **line** Line object.

11.4.1.4 virtual void PassengerInformationObserver::onServiceUpdated(Service service)

This function is called back when a new service is added or an existing service is updated in the system.

PARAMETERS

- ▶ **service** Service object.

11.4.1.5 virtual void PassengerInformationObserver::onServiceRemoved(Service service)

This function is called back when an existing service is removed from the system.

PARAMETERS

- **service** Service object.

11.4.1.6 virtual void PassengerInformationObserver::onVehicleUpdated(Vehicle vehicle)

This function is called back when a new vehicle is defined or an existing vehicle is updated in the system.

PARAMETERS

- **vehicle** Vehicle object.

11.4.1.7 virtual void PassengerInformationObserver::onVehicleRemoved(Vehicle vehicle)

This function is called back when an existing vehicle is removed from the system.

PARAMETERS

- **vehicle** Vehicle object.

11.4.1.8 virtual void PassengerInformationObserver::onStationUpdated(Station station)

This function is called back when a new station is defined or an existing station is updated in the system.

PARAMETERS

- **station** Station object.

11.4.1.9 virtual void PassengerInformationObserver::onStationRemoved(Station station)

This function is called back when an existing station is removed from the system.

PARAMETERS

- **station** Station object.

11.4.1.10 virtual void PassengerInformationObserver::onPlatformInfoUpdated(PlatformInfo platform)

This function is called back when a new platform is defined or an existing platform is updated in the system.

PARAMETERS

- **platform** PlatformInfo object.

11.4.1.11 virtual void PassengerInformationObserver::onPlatformInfoRemoved(PlatformInfo platform)

This function is called back when an existing platform is removed from the system.

PARAMETERS

- **platform** PlatformInfo object.

11.4.1.12 virtual void PassengerInformationObserver::onTripUpdated(Trip trip)

This function is called back when a new trip is defined or an existing trip is updated in the system.

PARAMETERS

- **trip** Trip object.

11.4.1.13 virtual void PassengerInformationObserver::onTripRemoved(Trip trip)

This function is called back when an existing trip is removed from the system.

PARAMETERS

- **trip** Trip object.

11.5 NamedId Class Reference

NamedId is a base class for objects that require an unique id, name and abbreviation. The NamedId class is the base class for the Vehicle, Station and PlatformInfo classes.

The "set" functions do not take place instantly, rather a change request is sent to the server. If the server accepts the update the change will take place. The change is propagated to the clients and the instance will get the new value.

The PassengerInformationObserver objects registered with the PassengerInformationServer will be notified of the change.

NOTE: Systems where the PIS system is interfacing with a signalling interface cannot use the "set" methods as the information is received directly from the signalling interface.

11.5.1 Constructor & Destructor Documentation

11.5.1.1 NamedId::NamedId(int id)

It is the constructor which initialises the unique Id of the object.

11.5.2 Member Function Documentation

11.5.2.1 int NamedId::getId()

Returns the numeric ID.

RETURNS

- ▶Returns the unique ID.

11.5.2.2 std::string NamedId::getName()

Returns the name of the object.

RETURNS

- ▶Returns the name of the object.

11.5.2.3 std::string NamedId::getAbrv()

Returns an abbreviated name of the object, where available in the information source. If abbreviated name is not set, returns the name of the object, identical to getName().

RETURNS

- ▶Returns the abbreviation of the object. If the object has no abbreviated name, an empty string is returned.

11.5.2.4 void NamedId::setName(const std::string &name)

Allows an application to set the name of the object.

PARAMETERS

- ▶**name** Name that needs to be set for the object.

11.5.2.5 void NamedId::setAbrv(const std::string &abbreviatedName)

Sets the abbreviation of the object.

Allows an application to set an abbreviated name for the object.

PARAMETERS

- ▶**abbreviatedName** Abbreviated name to set for the object

11.6 TripStop Class Reference

TripStop objects include information for each stop in a trip. This information includes the station and the platform within that station.

The TripStop represents a location where a scheduled trip can stop. Examples are particular platforms of a train station or stands of a bus stop.

A trip defines a particular stopping pattern, which is typically repeated over a number of days as specific services.

When a trip is defined by transport authorities, each of the stops has two scheduled time-offsets expressed in seconds, which define how long after trip start the vehicle is expected to arrive at, and depart from this stop.

11.6.1 Constructor & Destructor Documentation

11.6.1.1 TripStop::TripStop(int id)

Initialises the unique Id of a trip stop.

11.6.1.2 TripStop::TripStop(int id, int platformId, bool isStopping, long arrivalTimeOffset, long departureTimeOffset)

Initialises all attributes of a trip stop.

PARAMETERS

- ▶ **id** unique identifier of the TripStop object
- ▶ **platformId** Id of PlatformInfo object associated with the TripStop
- ▶ **isStopping** Indicates whether the Trip stops on the specified Platform
- ▶ **arrivalTimeOffset** Arrival time offset in seconds from Service startTime
- ▶ **departureTimeOffset** Departure time offset in seconds from Service startTime

11.6.2 Member Function Documentation

11.6.2.1 int TripStop::getTripId()

Returns the trip Id to which this stop belongs

RETURNS

- ▶ **Trip Id** to which this stop belongs

11.6.2.2 int TripStop::getPlatformId()

Returns the id of platform where vehicle stops or passes.

RETURNS

- **Platform Id T**he id of the platform where the vehicle stoppes or passes

11.6.2.3 bool TripStop::isStopping()

Indicates if the vehicle stops at the station or passes through. Elective stops are currently not supported.

RETURNS

- Returns true if the vehicle stops at the station; false if it passes through

11.6.2.4 long TripStop::getSchArvOffset()

Returns the arrival time offset in seconds from startTime

RETURNS

- Arrive time offset in seconds from startTime

11.6.2.5 long TripStop::getSchDepOffset()

Returns the departure time offset in seconds from startTime

RETURNS

- Returns the time offset in seconds from startTime

11.7 Trip Class Reference

The Trip object represents a stopping pattern. A stopping pattern is an ordered list of TripStops representing the locations that a Service will stop when executing a Trip. Trips are typically repeated multiple times over a number of days as Services.

Trips are generally defined in the transit authority's timetable system.

11.7.1 Public Types

- enum **Direction**
  - DESCENDING
  - ASCENDING

A datatype which shows which direction the trip is planned for. Direction typically indicates:

DESCENDING, or DOWN indicates the vehicle will be moving away from a Central station

ASCENDING, or UP indicates the vehicle will be moving towards a Central station.

- enum **Priority**
  - ALL_STOPS
  - LIMITED_STOPS
  - EXPRESS
  - PRIORITY_MAX

A datatype which shows what is the stopping pattern for the trip over a line.

11.7.2 Constructor & Destructor Documentation

11.7.2.1 Trip::Trip(int id, Direction direction, Priority priority)

Initialises all attributes of a Trip.

PARAMETERS

▶ **id** unique identifier of the Trip object

▶ **direction** Direction of Trip, as defined in the definition of the Direction enumerated type.

▶ **priority** Priority of Trip, as defined in the definition of the Priority enumerated type.

11.7.3 Member Enumeration Documentation

11.7.3.1 enum Trip::Direction

A datatype which shows which direction the trip is planned for.

ENUMERATION

- ▶ **DESCENDING** the trip moves away from the central location
- ▶ **ASCENDING** the trip moves towards the central location

11.7.3.2 enum Trip::Priority

A datatype which shows what is the stopping pattern for the trip over a line.

ENUMERATION

- ▶ **ALL_STOPS** vehicle stops in all the stations
- ▶ **LIMITED_STOPS** vehicle will not stop in a few stations in the line
- ▶ **EXPRESS** vehicle only stios in a few stations in the line

11.7.4 Member Function Documentation

11.7.4.1 int Trip::getLineId()

Returns the line Id to which the trip belongs

RETURNS

- ▶ Returns the unique line identifier.

11.7.4.2 Direction Trip::getDirection()

Returns the direction in which the trip is planned for

RETURNS

- ▶ Returns a variable of type Direction specifying the direction of the trip

11.7.4.3 Priority Trip::getPriority()

Returns the priority (express vs all stops) of the vehicle on the trip

RETURNS

- ▶Returns the stopping pattern of the vehicle on the trip

11.7.4.4 std::string Trip::getPriorityText()

Returns the textual representation of the priority (express vs all stops)

RETURNS

- ▶Returns priority in textual representation

11.7.4.5 std:string Trip::getStopsListAsString()

Returns the list of all the stop ids as a comma separated string.

RETURNS

- ▶Returns the list of all the stop ids as a comma separated string.

11.7.4.6 void Trip::setStops(std::vector<int> tripStopIds)

Returns the list of all the stop ids as a comma separated string.

PARAMETERS

- ▶**tripStopIds** List of TripStop ids comprising the Trip.

11.8 ServiceStop Class Reference

The ServiceStop object represents a TripStop scheduled for Service. The ServiceStop has dynamic arrival and departure time information which is updated in real-time by the system as required.

Dynamic updates may be received by the NetSpire system from the signalling system, or through location information provided by equipment on the vehicle.

Each service is an instance of a trip with a specific start time, and arrival/departure times defined for each stop. The delays for each stop can be controlled individually.

11.8.1 Constructor & Destructor Documentation

11.8.1.1 ServiceStop::ServiceStop(int id, int serviceId, int tripStopId)

Initialises all attributes of a service stop.

PARAMETERS

- ▶**id** unique identifier of the object
- ▶**serviceId** Id of Service object to which this ServiceStop belongs
- ▶**tripStopId** Id of TripStop object from which this ServiceStop was derived

11.8.2 Member Function Documentation

11.8.2.1 int getServiceId()

Returns the unique service identification to which this ServiceStop belongs.

RETURNS

- ▶Returns the service identification to the owning service

11.8.2.2 int getTripStopId()

Returns the TripStop Id from which this ServiceStop was derived.

RETURNS

- ▶Returns the trip stop Id

11.8.2.3 long ServiceStop::getServiceStartTime()

Returns the time when the vehicle started the trip in the first station

RETURNS

- Returns the time when the vehicle started the trip in the first station

11.8.2.4 long ServiceStop::getArvDelay()

Returns the amount of delay expected for the vehicle to arrive the station

RETURNS

- Returns the amount of delay expected for the vehicle to arrive the station

11.8.2.5 long ServiceStop::getDepDelay()

Returns the amount of delay expected for the vehicle to depart the station

RETURNS

- Returns the amount of delay expected for the vehicle to depart the station

11.8.2.6 void ServiceStop::setArvDelay(long delay)

Sets the amount of delay expected for the vehicle to arrive at the station.

PARAMETERS

- **delay** The amount of delay in seconds expected for the vehicle to arrive the station

11.8.2.7 void ServiceStop::setDepDelay(long delay)

Sets the amount of delay expected for the vehicle to depart the station

PARAMETERS

- **delay** The amount of delay in seconds expected for the vehicle to depart the station

11.8.2.8 long ServiceStop::getArvTime()

Returns the scheduled arrival time

RETURNS

- Returns scheduled arrival time. This is calculated as startTime + arvOffset

11.8.2.9 long ServiceStop::getDepTime()

Returns the scheduled departure time

RETURNS

- ▶Returns scheduled departure time. This is calculated as startTime + depOffset

11.9 Service Class Reference

A service is a vehicle servicing a specific trip. A service is created for each train on a scheduled or dynamically created trip. The service start time includes the date and time the service is starting/has started its run.

11.9.1 Public Types

- enum **State**
  - NOT_STARTED
  - RUNNING
  - APPROACHING_STATION
  - IN_STATION
  - BOARDING
  - DEPARTING_STATION
  - DEPARTED_STATION
  - TERMINATED
  - CANCELLED

A data type defining the state of the service.

11.9.2 Constructor & Destructor Documentation

11.9.2.1 Service::Service(int id, int tripId, long startTime)

Initialises all attributes of a service.

PARAMETERS

- ▶**id** unique identifier of the Service object
- ▶**tripId** Id of Trip object associated with the TripStop
- ▶**startTime** Service time offset in seconds from Jan 1, 1970.

11.9.3 Member Enumeration Documentation

11.9.3.1 enum Service::State

A data type definig the state of the service

ENUMERATION

- ▶**NOT_STARTED** The service has not started yet.
- ▶**RUNNING** The service is started and running between two stations.

- ▶ **APPROACHING_STATION** The train in this service is approaching a station.
- ▶ **IN_STATION** The train in this service is in a station.
- ▶ **BOARDING** The train in this service is boarding passengers.
- ▶ **DEPARTING_STATION** The train in this service is about to leave the station.
- ▶ **DEPARTED_STATION** The train in this service has left the station.
- ▶ **TERMINATED** The service has has reached the destination and terminated.
- ▶ **CANCELLED** The service has been cancelled.

11.9.4 Member Function Documentation

11.9.4.1 int getTripId()

Returns the unique trip identification based on which the service is made

RETURNS

- ▶ Returns the trip based on which the service is made

11.9.4.2 long Service::getStartTime()

Returns the arrival time offset in seconds from start time.

RETURNS

- ▶ Returns the arrival time offset in seconds from start time.

11.9.4.3 int Service::getVehicleId()

Returns the unique identification of the vehicle performing the service.

RETURNS

- ▶ Returns the vehicle Id

11.9.4.4 State Service::getState()

Returns the state of the service.

RETURNS

- ▶ Returns the state of the service

11.9.4.5 std::string Service::getStateText()

Returns the textual representation of the current state of the service.

RETURNS

- ▶Returns state in textual format

11.9.4.6 void Service::setStops(const std::list<ServiceStop>& stops)

Sets the stops that are members of this service. The ServiceStop entries must start with the first stop that is to be displayed or announced in the system, followed by each subsequent stop. The last entry in the list is displayed as the destination of this service.

11.9.4.7 std::vector<ServiceStop> Service::getStopsSchedule()

Returns a list of stations with their timing information as ServiceStop objects.

RETURNS

- ▶Returns list of stops scheduled for this service.

11.9.4.8 void Service::getStopsSchedule(std::vector<ServiceStop> stops)

Sets a list of stations with their timing information included in the service schedule.

PARAMETERS

- ▶**stops** List of stops scheduled for this service.

11.9.4.9 std::string Service::getFollowingStopsListAsStopsSchedule()

Returns a list of station Ids as comma separated string.

RETURNS

- ▶Returns list of stops scheduled for this service.

11.10 Vehicle Class Reference

The Vehicle objects represent and track vehicles in the system like trains, busses, trams and ferries.

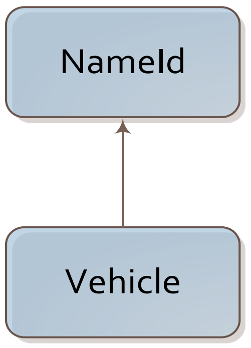


Figure 9 - Vehicle Inheritance Diagram

11.10.1 Public Types

- enum **Direction**
  - DESCENDING
  - ASCENDING

  A datatype defining the direction of the vehicle's movement

- enum **State**
  - OUT_OF_SERVICE
  - IN_SERVICE
  - SWITCHING_SERVICE
  - STATE_MAX

  A datatype defining the state of the vehicle

- enum **DoorSide**
  - NONE,
  - SIDE_1,
  - SIDE_2,
  - BOTH
  - DOOR_SIDE_MAX

  A datatype defining the side where doors are/will be opened in the current/next station

11.10.2 Detailed Description

It is a class representing vehicles in the system. It could be trains, buses or ferries,

11.10.3 Member Enumeration Documentation

11.10.3.1 enum Trip::Direction

A datatype which shows which direction the vehicle travels

ENUMERATION

- ▶ **DESCENDING** the vehicle goes away from the central location
- ▶ **ASCENDING** the vehicle goes towards the central location

11.10.3.2 enum Vehicle::State

A datatype defining the state of the vehicle.

ENUMERATION

- ▶ **OUT_OF_SERVICE** The vehicle is out of service.
- ▶ **IN_SERVICE** The vehicle is currently running a service.
- ▶ **SWITCHING_SERVICE** The vehicle has finished a service and is being assigned to another.

11.10.3.3 enum Vehicle::DoorSide

A datatype defining which door will be open in the next station.

ENUMERATION

- ▶ **NONE** The doors are not opened/supposed to open.
- ▶ **SIDE_1** The doors on side 1 are expected to be opened (Left/Right depends on vehicle's moving direction).
- ▶ **SIDE_2** The doors on side 2 are expected to be opened (Left/Right depends on vehicle's moving direction).
- ▶ **BOTH** Both of the doors are expected to be opened.

11.10.4 Member Function Documentation

11.10.4.1 int Vehicle::getNumCars()

Returns the number of cars in the vehicle

RETURNS

- Returns the number of cars in the vehicle

11.10.4.2 Service Vehicle::getCurrentService() const throw (std::out_of_range)

Returns the service that the vehicle is currently serving

RETURNS

- Returns the service that the vehicle is currently serving

EXCEPTIONS

- **std::out_of_range** if no service is scheduled for this train

11.10.4.3 std::vector<Service> Vehicle::getServices()

Returns the list of all the services scheduled to be served by this vehicle

RETURNS

- Returns the list of all the services scheduled to be served by this vehicle

11.10.4.4 Direction Vehicle::getDirection()

Returns the direction in which vehicle moves

RETURNS

- Returns a variable of type Direction specifying the direction of train movement

11.10.4.5 LocationId Vehicle::getCurrentLocation()

Returns the current location of the vehicle

RETURNS

- Returns the location within network position system where train currently is

11.10.4.6 State Vehicle::getState()

Returns the state of the vehicle

RETURNS

Returns the state of the vehicle

11.10.4.7 std::string Vehicle::getStateText()

Returns textual representation of the state of the vehicle

RETURNS

Returns textual representation of the state of the vehicle

11.10.4.8 DoorSide Vehicle::getOpeningDoors()

Returns the DoorSide that will open at next/current stop

RETURNS

- Returns the side whose door will open

11.10.4.9 std::string Vehicle::getOpeningDoorsText()

Returns the textual representation of the DoorSide that will open at next/current stop

RETURNS

- Returns textual representation of the side whose door will open

11.10.4.10 DoorSide Vehicle::getOpenedDoors()

Returns the DoorSide that is currently opened

RETURNS

- Returns the side whose door is opened

11.10.4.11 std::string Vehicle::getOpenedDoorsText()

Returns the textual representation of the DoorSide that is currently opened.

RETURNS

- Returns textual representation of the side whose door is opened

11.10.4.12 void Vehicle::createService(long tripId, long startTime)

This function allows the operator to create a new service for the vehicle.

PARAMETERS

- **tripId** id for the new trip created for the service
- **startTime** time offset in seconds indicating the start time of the trip.

RETURNS

None

11.10.4.13 void Vehicle::deleteService(int serviceId)

This function allows deletes an existing service from the system.

PARAMETERS

- **serviceId** id of the service that needs to be removed

RETURNS

None

11.10.4.14 void Vehicle::clearServices()

This function removes all services that are associated for the vehicle.

PARAMETERS

None

RETURNS

None

11.10.4.15 void Vehicle::replaceService(long tripId, long startTime)

This function removes all services associated with the vehicle and adds a new service.

PARAMETERS

- **tripId** id for the new trip created for the service
- **startTime** time offset in seconds indicating the start time of the trip.

RETURNS

None

11.10.4.16 void Vehicle::setNumCars(int cars)

Sets the number of cars in the vehicle.

PARAMETERS

- ***cars*** Number of cars in the vehicle

11.10.4.17 void Vehicle::setCurrentServiceId(int currentServiceId)

Sets the service that the vehicle is currently serving.

PARAMETERS

- ***service*** Service that the vehicle is currently serving

11.10.4.18 void Vehicle::setServices(std::vector<Service>& serviceList)

Sets the list of all the services scheduled to be served by this vehicle.

PARAMETERS

- ***serviceList*** All services that the vehicle is scheduled to be associated with

11.10.4.19 void Vehicle::setDirection(Direction direction)

Sets the vehicle direction.

PARAMETERS

- ***direction*** Specifies direction of train movement

11.10.4.20 void Vehicle::setCurrentLocation(LocationId location)

Sets the current location of the vehicle.

PARAMETERS

- ***direction*** Location of vehicle within the network position system.

11.10.4.21 void Vehicle::setState(State state)

Assigns the state of the vehicle.

PARAMETERS

- ***state*** State of vehicle.

11.11 PlatformInfo Class Reference

The PlatformInfo objects represent specification locations within stop where a vehicle can stop. These locations may be platforms on a train station or stands in a bus terminal.

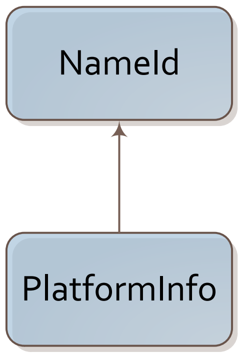


Figure 10 - PlatformInfo Inheritance Diagram

11.11.1 Public Types

- enum **State**
  - NO_VEHICLE_SCHEDULED
  - VEHICLE_SCHEDULED
  - VEHICLE_APPROACHING
  - VEHICLE_ON_PLATFORM
  - VEHICLE_DOES_NOT_STOP
  - VEHICLE_TERMINATED
  - VEHICLE_BOARDING
  - VEHICLE_DEPARTING
  - VEHICLE_DEPARTED

The datatype defining the state of the platform.

11.11.2 Member Enumeration Documentation

11.11.2.1 enum PlatformInfo::State

The datatype defining the state of the platform.

ENUMERATION

▶**NO_VEHICLE_SCHEDULED** No Vehicle is scheduled to pass this platform.

▶**VEHICLE_SCHEDULED** Some vehicles are scheduled to pass this platform but they are not approaching.

▶**VEHICLE_APPROACHING** Next vehicle is approaching the platform.

- **VEHICLE_ON_PLATFORM** The vehicle is on platform.
- **VEHICLE_DOES_NOT_STOP** When the train doesn't stop the state transitions to **DOES_NOT_STOP** instead of ON_PLATFORM and then to DEPARTED.
- **VEHICLE_TERMINATED** It is the last stop in the serivce and train terminates here.
- **VEHICLE_BOARDING** The vehicle is boarding passengers.
- **VEHICLE_DEPARTING** The vehicle is about to depart.
- **VEHICLE_DEPARTED** The vehicle departed the platform.

11.11.3 Constructor & Destructor Documentation

11.11.3.1 PlatformInfo::PlatformInfo(int id)

PlatformInfo constructor - requires a numeric id.

PARAMETERS

- **id** Numberic Id for the platform

11.11.4 Member Function Documentation

11.11.4.1 int PlatformInfo::getStationId()

Allows an application to get the unique station id where this platform reside in

RETURNS

- **station** Returns the station Id

11.11.4.2 LocationId PlatformInfo::getLocation()

Returns the location which represents where the platform is in the network position system

RETURNS

- Returns the location within network position system where the platform is

11.11.4.3 State PlatformInfo::getState()

Returns the state of the platform

RETURNS

- **State** current state of the platform

11.11.4.4 std::string PlatformInfo::getStateText()

Returns the textual representation of the state of the platform.

RETURNS

- ▶ Returns state text

11.11.4.5 std::list<ServiceStop> PlatformInfo::getPassers()

Returns a list of all the serviceStops representing this platform in a scheduled service

RETURNS

- ▶ **std::list<ServiceStop>** Returns a list of all the serviceStops representing this platform in a scheduled service

11.11.4.6 void PlatformInfo::setLocation(LocationId& location)

Sets the location which represents where the platform is in the network position system

PARAMETERS

- ▶ **location** Location of platform within network position system

11.11.4.7 void PlatformInfo::setState(State state)

Sets the state of the platform.

PARAMETERS

- ▶ **state** current state of the platform

11.11.4.8 void PlatformInfo::setPassers(std::list<ServiceStop> serviceStops)

Sets a list of all scheduled serviceStops servicing this platform.

PARAMETERS

serviceStops List of all the serviceStops servicing this platform. Each entry must contain a ServiceStop entry where ServiceStop:: getPlatformId () is the ID of this platform. Each entry must be part of a valid Service registered in the system.

11.11.4.9 void PlatformInfo::setPassers(std::list<Service> services)

Sets a list of all scheduled services servicing this platform.

PARAMETERS

services List of all the services servicing this platform. Each entry must contain a list of ServiceStops, where the first ServiceStop entry has ServiceStop:: getPlatformId () set to the ID of this platform. The Services specified using this method do not need to registered in the system prior to use.

11.12 Station Class Reference

The Station object represents a location where a vehicle can stop. These can be train stations, bus stops, tram stops and ferry wharf.

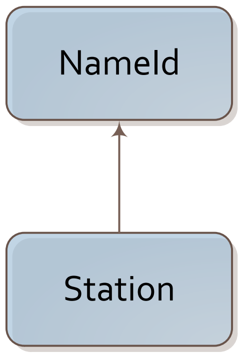


Figure 11 - Station Inheritance Diagram

11.12.1 Constructor & Destructor Documentation

11.12.1.1 Station::Station ()

Default constructor of the object.

11.12.1.2 Station::Station (int id, const std::string &name, bool isMajor)

Constructor initialising base class and static attributes of a station.

PARAMETERS

- ▶ **id** Station id
- ▶ **name** Station name
- ▶ **isMajor** True for major stations. This information is used in some systems where all the stations in a trip are not announced and only the major ones are announced.

11.12.2 Member Function Documentation

11.12.2.1 bool Station::isMajor()

Returns indication of whether this station is a major station. This information is used in some systems where all the stations in a trip are not announced and only the major ones are announced.

RETURNS

- Returns true if the station is a major one

11.12.2.2 std::map<int,PlatformInfo> Station::getListOfPlatforms()

Returns list of all the platforms within this station

RETURNS

- Returns list of all the platforms exist within this station

11.12.2.3 void Station::setListOfPlatforms(std::map<int,PlatformInfo> platforms)

Assigns platforms belonging to this station. The stationed reference available via PlatformInfo::getStationId() will automatically be assigned to reflect the relationship when platforms are assigned to a station.

PARAMETERS

- **platforms** Platforms within this station.

11.12.2.4 void Station::setPlatform(int platformID, const PlatformInfo& platform)

Assign a platform belonging to this station. The stationed reference available via PlatformInfo::getStationId() will automatically be assigned to reflect the relationship when platforms are assigned to a station.

PARAMETERS

- **platformId** numberic ID of the platform
- **platform** Platform within this station.

11.12.2.5 PlatformInfo Station::getPlatform(int platformId) const throw (std::out_of_range)

Returns the platform information of a specific platform

PARAMETERS

- **platformId** numberic ID of the platform whose information should be returned

RETURNS

- Returns the platform information of a specific platform

EXCEPTIONS

- **std::out_of_range** If a platform with that numeric id doesn't exist in this station

12 Sample Applications & Code Excerpts

12.1 About This Section

This section provides an overview of the sample applications provided with the NetSpire SDK Library. The sample applications may be used as a guide for developers interfacing with the system.

12.2 Sample Applications

The Sample Application Summary table provides an overview of the sample applications included in the standard NetSpire SDK distribution.

Sample applications are provided which demonstrate the use of synchronous polling and event based operation.

An Example application developed in each supported languages is also provided.

In addition to the sample applications, code excerpts showing how to invoke a range of common functions is provided.

12.3 Sample Application Summary

Application	Description	API	Lang.	Section
monitorDeviceStates.cpp	Retrieves and displays real time status information for all subordinate devices in a NetSpire system.	POLLING + EVENT	C++	Section 16.1
monitorDeviceStates.java	Retrieves and displays real time status information for all subordinate devices in a NetSpire system	POLLING + EVENT	JAVA	Section 16.2
monitorDeviceStates.cs	Retrieves and displays real time status information for all subordinate devices in a NetSpire system	POLLING + EVENT	C#	Section 16.3
updateServiceInformation.cpp	Sets the current information on Vehicles, Stations, Platforms and Services for use in automatic message generation	POLLING	C++	Section 16.4

Table 56 - Sample Application Summary

12.4 Sample Code Excerpt Overview – General Functions

Name	Description	API	Lang.	Section
Make Live PA Announcement	Make PA announcement from a pre-defined input (Analog or Network Paging Station)	POLLING	C#	Section 12.9
Make DVA announcement	Make DVA announcement to all available PaSink (on all devices simultaneously)	POLLING	C#	Section 12.10
Show Message on Display	Display a message on a LCD screen	POLLING	C#	Section 12.11
Set Gain Levels on Input	Set gain levels for a specificed input source	POLLING	C#	Section 12.12
Set Gain Levels on Output	Set absolute gain levels for all audio outputs (sinks) in a zone	POLLING	C#	Section 12.13
Show audio activity status	Display audio zone activity status for all zones	POLLING	C#	Section 12.14
Show device states	Retrieve Device State of all devices	POLLING	C#	Section 12.15
Set dynamic config	Change dynamic global configuration value	POLLING	C#	Section 12.16
Make telephony call	Make telephony call between two specific parties	POLLING	C#	Section 12.17
Show active telephony calls	Show details for all active telephony calls in the system.	POLLING	C#	Section 12.18
Accept a telephone call	Accept a telephony call on a specific terminal	POLLING	C#	Section 12.19
Hold a telephone call	Hold a call on a specific terminal	POLLING	C#	Section 12.20
Resume a held telephony call	Resume a call on a specific terminal	POLLING	C#	Section 12.21
Transfor a call	Blind transfer a call to another terminal.	POLLING	C#	Section 12.22
Upload Dictionary	Upload new audio segments to the centrally managed dictionary	POLLING	C#	Section 12.23
Check Dictionary Update Status	Check if dictionary is consistent version level at all locations	POLLING	C#	Section 12.24
Display Dictionary	Retrieve the contents of the dictionary	POLLING	C#	Section 12.25
Check firmware version	Check if all devices are running the same firmware	POLLING	C#	Section 12.26
Read Digital Input	Read the value of a device's digital input	POLLING	C#	Section 12.27
Set Digital Output	Set the value of a device's digital output	POLLING	C#	Section 12.28

Table 57 - Sample Code Excerpt Summary

12.5 Sample Application Configuration

All NetSpire applications communicate with a communications exchange and must specify an IP address for at least one of these servers. By default, the network port number and transport protocol for the server is set to 20000 and TCP and will typically not need to be changed. The application will automatically select a network interface in accordance with the routing configuration for the system. In the example applications, the server addresses are specified in defines in the program and need to be changed to match the specific environment.

12.6 monitorDeviceStates C++

Sample application file:

monitorDeviceStates.cpp

Description:

This application retrieves and displays real time state information for devices in a NetSpire system. The application can operate as a synchronous polling or asynchronous event driven application by setting the pre-define *gUsePolling* to "true" for synchronous operation.

The application performs the following:

- ▶ Initialises the NetSpire Framework.
- ▶ If operating in asynchronous mode, registers observers for all state events.
- ▶ Establishes a connection to a NetSpire server.
- ▶ If operating in polling mode, waits for a connection to form and then loops periodically to display device states.
- ▶ If operating in event driven mode, event updates will show updated information as device state updates are received.

Compilation and Configuration:

This application must first be compiled and linked prior to being run. Project and make files are provided for both Linux and Windows environments. No specifcal configuration other than setting the server addresses and mode of polling is needed.

Execution

No command line arguments are required and the application can be run directly in a console or terminal window.

Prerequisites:

This application requires a NetSpire communication server to be running and ideally one or more NetSpire devices to be present and configured to use the server.

To view status changes, the hardware being monitored will need to change state. For telephones or intercoms, this will mean changing the hook state or initiating a call. For amplifers, this will mean making an announcement.

12.7 monitorDeviceStates Java

Sample application file:

monitorDeviceStates.java

Description:

This application is similar to the monitorDeviceState C++ application as described in the previous section and only aspects specific to the Java version of the application will be referenced here.

Compilation and Configuration:

This application must first be compiled and linked prior to being run. Project and make files are provided for both Linux and Windows environments. No specifcal configuration other than setting the server addresses and mode of polling is needed.

To compile on the command line in Microsoft Windows, the following command line can be used.

```
javac.exe -cp as.jar monitorDeviceStates.java
```

Execution

No command line arguments are required and the application can be run directly in a console or terminal window using the java interpreter. An example line for executing the application under windows is shown below:

```
java.exe" -cp as.jar;. monitorDeviceStates
```

Prerequisites:

This application requires Java SE 7 or higher as well as the JDE for compilation.

This application requires a NetSpire communication server to be running and ideally one or more NetSpire devices to be present and configured to use the server.

To view status changes, the hardware being monitored will need to change state. For telephones or intercoms, this will mean changing the hook state or initiating a call. For amplifers, this will mean making an announcement.

12.8 monitorDeviceStates C#

Sample application file:

monitorDeviceStates.cs

Description:

This application is similar to the monitorDeviceState C++ application as described in the previous section and only aspects specific to the C# version of the application will be referenced here.

Compilation and Configuration:

This application must first be compiled and linked prior to being run. Project and make files are provided for the Windows environments. No specifcal configuration other than setting the server addresses and mode of polling is needed.

To compile the application use the Microsoft Visual Studio project files provided.

Execution

No command line arguments are required and the application can be run directly in a console or terminal window using the .NET runtime environment.

Prerequisites:

This application requires Microsoft .NET version 4 or higher.

This application requires a NetSpire communication server to be running and ideally one or more NetSpire devices to be present and configured to use the server.

To view status changes, the hardware being monitored will need to change state. For telephones or intercoms, this will mean changing the hook state or initiating a call. For amplifers, this will mean making an announcement.

12.9 Make PA announcement from an Analog input/IPPA input

```
// get list of available PA sources and find PaSource called ‘Central_MIC_1’.
Initiate a PA announcement from this source to
// all available zones in the system. The PaSource may be any source type as
defined during system configuration – eg Analog Input or IP Paging Station.

int paSourceId = 0;
netspire.PAController paC = gAudioServer.getPAController();
netspire.PaSourceArray paSourceArray = new netspire.PaSourceArray();
paSourceArray = paC.getPaSources();
for (int i = 0; i < paSourceArray.Count; i++)
{
       if (paSourceArray.ElementAt(i).label == "Central_MIC_1")
       {
              paSourceId = paSourceArray.ElementAt(i).id;
              break;
       }
}
// attach PaSource (take control of the Microphone for making announcement)
paC.attachPaSource(paSourceID);
// attach PaSink to PaSource. PaSink Ids can be retrived using the function
getAudioSinks()
netspire.PaSinkArray availableZonesFromAS = pac.getPaSinks();
for (int i = 0; i < availableZonesFromAS.Count; i++)
{
       paC.attachPaSink(sourceID, availableZonesFromAS.ElementAt(i).id,
netspire.PaSink.Mode.ADD_TO_EXISTING_SET);
}
// create SW trigger to allow SDK to act on the PaSource
#define PRIORITY_LEVEL 100
int swTriggerId = paC.createSwPaTrigger(sourceID, PRIORITY_LEVEL);
// start PA Announcement to the attached PaSinks
paC.activateSwPaTrigger(swTriggerId);
// sleep for 2 seconds to allow for fixed duration PA for 2 seconds
sleepMS(2000);
// stop PA Announcement
paC.deactivateSwPaTrigger(swTriggerId);
```

12.10 Make DVA announcement to all available PaSinks

```
// Following example shows how to initiate a digital voice announcement to all
available zones
// in the system.
// A predefined dictionary segment with id = 1234 is played to all available
zones. Function ‘getDictionaryItems()’ can be used to retrieve the full list of
dictionary segments.

netspire.PaSinkArray sinkListFromAS = pac.getPaSinks();
netspire.PaSinkArray targetSinkArray = sinkListFromAS;

// associate visualDevices (none for this example)
netspire.StringArray visualDevicesStringArray = new netspire.StringArray();
// set Gain for outputs in all target zones for the duration of this DVA
announcement.
netspire.Gain gain = new netspire.Gain();
gain.setLevel(18.00);
```

```
// Specify a static dictionary ID of 1234 for this message.
// Note that to determine available dictionary items via introspection, use
// the function getDictionaryItems() to return a full list.

int dictionaryId = 1234;
netspire.NumberArray dvaItems = new netspire.NumberArray();
dvaItems.Add(dictionaryId); // Note that dictionary items are always numeric

// Do not include any visual display component in this example
string visualText = null;

// call function to start playback of dictionary item to all PaSinks
#define NOT_USED 0
#define OVERIDING_EXISTING_ANNOUNCEMENT false
#define RESUME_ON_INTERRUPT false
uint messageId = paController.playMessage(targetSinkArray,
visualDevicesStringArray, gain, dvaItems, visualText, RESUME_ON_INTERRUPT,
OVERIDING_EXISTING_ANNOUNCEMENT, NOT_USED, NOT_USED, NOT_USED);
```

12.11 Display a message on a LCD screen

```
// Following example shows how to update all available LCD screen with a new
message.

// generate a list of selected sinkIds. No zones need to be selected for
// this example as we are only interested in updating the LCD screen.


#define NOT_USED 0
#define OVERIDING_EXISTING_ANNOUNCEMENT false
#define RESUME_ON_INTERRUPT false

netspire.PaSinkArray targetSinkArray = new netspire.PaSinkArray();
// set visualDevices
netspire.StringArray visualDevicesStringArray = new netspire.StringArray();
// set Gain (n/a for this scenario)
netspire.Gain gain = new netspire.Gain();
// generate a list of DVA dictionary items (n/a for this scenario)
netspire.NumberArray dvaItems = new netspire.NumberArray();
// Set visual Message for display
string visualText = "This is a test message to LCD screen";
// call function to update the message on the LCD screen
uint messageId = paController.playMessage(targetSinkArray,
visualDevicesStringArray, gain, dvaItems, visualText, RESUME_ON_INTERRUPT,
OVERIDING_EXISTING_ANNOUNCEMENT, NOT_USED, NOT_USED, NOT_USED);
```

12.12 Set gain levels for an input source

```
// Applies an input gain level to a specific dB at the specified input source.
// Iterates through available sources, finds input to match specified source ID
and sets gain.

// source id of input to set.
#define CENTRAL_MIC_PA_SOURCE_ID 210000201

double absoluteGainLevel = 4.00;  // gain level in dB to be set
netspire.PAController paC = gAudioServer.getPAController();      // an instance
of PAController
```

```
// retrieve available PaSources
netspire.PaSourceArray paSourceArray = new netspire.PaSourceArray();
paSourceArray = paC.getPaSources();

// Loop through all sources. Find the source ID of interest and and set new gain

for (int i = 0; i < paSourceArray.Count; i++)
{
	if (paSourceArray.ElementAt(i).id == CENTRAL_MIC_PA_SOURCE_ID)
	{
		netspire.PaSource paSource = paSourceArray.ElementAt(i);
		paSource.setGain(absoluteGainLevel, false);
	}
}
```

12.13 Set absolute gain levels for all audio outputs (sinks) in a zone

```
// Set output gain level for all output devices (audio sinks) in zone
'COMMON_AREA'
#define COMMON_AREA_ID 210000201
double absoluteGainLevel = -12.00;     // gain level in dB to be set
     // Retrieve list of all available PaSinks

     netspire.PaSinkArray paSinkArray = gAudioServer.getAudioSinks();
     // Loop through all PaSinks. Find the zone of interest and set the gain
     for (int i = 0; i < paSinkArray.Count; i++)
     {
      if (paSinkArray.ElementAt(i).id == COMMON_AREA_ID)
           {
            netspire.PaSink paSink = paSinkArray.ElementAt(i);
            paSink.setGain(new netspire.Gain(absoluteGainLevel), false);
           }
     }
```

12.14 Display audio zone activity status for all zones

```
      // Retrieve list of all available PaSink and display current activity status
      to the stdout.
netspire.PaSinkArray paSinkArray = pac.getPaSinks();
      // Loop through all PaSinks and get the state (IDLE, ACTIVE, UNKNOWN) and
      the announcementType (FILE_PLAY, VOIP_STREAMING, etc)
      for (int i = 0; i < paSinkArray.Count; i++)
      {
       int sinkId = paSinkArray.ElementAt(i).id;
       netspire.PaSink.State sinkState = paSinkArray.ElementAt(i).state;
             netspire.PaSink.AnnouncementType announType =
      paSinkArray.ElementAt(i).announType;
             Console.WriteLine("sinkState >" + sinkState + "<");
      }
```

12.15 Retrieve Device State of all devices

```
      // Retrieve list of all devices and display current device state to stdout.
      netspire.DeviceStateArray devStateArray = gAudioServer.getDeviceStates();
      // Loop though the list and get/check the device state. Device state can be
      IDLE, ACTIVE, COMMSFAULT, FAULTY, ISOLATED.
      for (short i = 0; i < devStateArray.Count; i++)
      {
             string deviceState =
      devStateArray.ElementAt(i).getStateText().ToString();
             Console.WriteLine("deviceState >" + deviceState + "<");
      }
```

12.16 Change dynamic global configuration value

```
      // Following example shows how to set the delay after PA before resuming
      DVA. The global configuration variable “DVA_Post_Announcement_Delay_MS” will
      be set to
      // 2 seconds and will apply globally over all devices in the system.

      netspire.KeyValueMap userDefinedDynamicConfig = new netspire.KeyValueMap();
      userDefinedDynamicConfig.Add("DVA_Post_Announcement_Delay_MS", "2000");
      gAudioServer.setDynamicConfiguration(userDefinedDynamicConfig);
```

12.17 Make new call between two specified parties

```
      // Following example shows how to make a new call from a specified end point
      to an arbitrary address The following example will create a call between
      extension 3011 and 4023.

      string aPartyCLI = "3011";  // Extension of A-party
      string bPartyCLI = "4023";  // Extension of B-party
      netspire.CallController caC = gAudioServer.getCallController();    // get an
      instance of CallController
      int callId = caC.createCall(aPartyCLI, bPartyCLI);           // request
      system to create a new call and return the call reference id
      if (callId > 0)
      {
             Console.WriteLine("Call created - callId >" + callId + "<");
      }
      else
      {
             Console.WriteLine("Failed to create call");
```

```
}
```

12.18 Show all active calls in the system

```
// Following example shows how to retrieve all active telephony calls in the
system and then iterate through them to extract information associated with
the call.

netspire.CallController caC = gAudioServer.getCallController();    // get an
instance of CallController
netspire.CallInfoArray callInfoArray = caC.getCalls();             // get
list of all calls
```

```
// iterate through all calls and extract information
for (short i = 0; i < callInfoArray.Count; i++)
{
// get call Id
    String callId = callInfoArray.ElementAt(i).callId.ToString();
// get call start time
    String callStartTime =
callInfoArray.ElementAt(i).callStartTime.ToString();
// get call duration
    String callDuration =
callInfoArray.ElementAt(i).callDuration.ToString();
// get call answer time
    String callAnswerTime =
callInfoArray.ElementAt(i).callAnswerTime.ToString();
// get a-party
    String callAPartyId =
callInfoArray.ElementAt(i).callAPartyId.ToString();         // get b-party
    String callBPartyId =
callInfoArray.ElementAt(i).callBPartyId.ToString();
// get list of target Ids
    String callTargetIds =
callInfoArray.ElementAt(i).callTargetIds.ToString();
// get current call state
    String callState = callInfoArray.ElementAt(i).callState.ToString();

// get the hang-up cause
    String callReleaseCause =
callInfoArray.ElementAt(i).callReleaseCause.ToString();
};
```

12.19 Accept a call at a specific terminal

```
// Following example shows how to accept an incoming call on terminal ID
4026 initiated by terminal ID 3011
netspire.CallController caC = gAudioServer.getCallController();    // get an
instance of CallController
netspire.CallInfoArray callInfoArray = caC.getCalls();             // get
list of all calls
// iterate through all calls and extract information
for (short i = 0; i < callInfoArray.Count; i++)
{
 if (callInfoArray.ElementAt(i).callAPartyId() == "4026")
      {
       int callId = callInfoArray.ElementAt(i).callId;
       caC.answerCallOnTerminal(callInfoArray.ElementAt(i).callId, "4026",
"3011");
       break;
      }
}
```

12.20 Hold Call on a specific terminal

```
// Following example shows how to put a call on hold. In this example,
extension 4026 will be put on hold. The value for callID has already been
set by the call observer call back function “onCallUpdate()”.
string bPartyCLI = "4026";  // Extension of B-Party

// get an instance of CallController
netspire.CallController caC = gAudioServer.getCallController();
```

```
caC.holdCall(callId, bPartyCLI);        // request system to put B-Party on
hold
```

12.21 Resume Call on a specific terminal

```
// Following example shows how to resume a call for b-party which is
currently on hold. The
// value for callID has already been set by the call observer call back
function “onCallUpdate()”.
string bPartyCLI = "4026";  // Extension of B-Party

// get an instance of CallController
netspire.CallController caC = gAudioServer.getCallController();
 caC.resumeCall(callId, bPartyCLI);           // request system to resume
call for B-Party
```

12.22 Blind transfer a call form one terminal to another.

```
// Following example shows how to transfer call to a new b-party. The
// value for callID has already been set by the call observer call back
function “onCallUpdate()”.
string bPartyCLI = "4026";  // Extension of B-Party
netspire.CallController caC = gAudioServer.getCallController();    // get an
instance of CallController
caC.transferCall(callId, bPartyCLI);                          // request
system to transfer A-Party to the new B-Party
```

12.23 Upload a new dictionary version

```
// Upload a new dictionary package that has been prepared in accordance with
the structure specified in the SDK document.
// The following function will upload the package to Communication Exchange
which will
// subsequently update //the dictionary on all relevant devices.
// The function will return a bool to indicate if the update of the
dictionary package was //successful. The function ‘getDeviceStates()’ should
be called to check that the dictionary version number for the device is
consistent with the version in the package.
bool ret = gAudioServer.updateDictionary("c:\\test\\newDictionary.zip");
```

12.24 Check if dictionary is consistent version level at all locations.

```
// Following example shows how to check if all the devices have synchronized
their dictionary to the Communication Exchange.
// The example will retrieve list of all available devices in the system and
then iterate
// through them to get the dictionary version.
// The dictionary version will indicate if the dictionary is synchronized
with the Communication Exchange.

netspire.DeviceStateArray devStateArray = gAudioServer.getDeviceStates();
for (short i = 0; i < devStateArray.Count; i++)
{
    string dictionaryVersion =
devStateArray.ElementAt(i).getDictionaryVersion().ToString();
}
```

12.25 Retrieve the contents of the dictionary

```
// Following example shows how to retrieve a list of all dictionary items
present in the system. Once the list is retrieved, loop
// through it to extract the dictionary item no and description for each
item.
netspire.DictionaryItemArray dictionaryItemList =
gAudioServer.getDictionaryItems();
for (int i = 0; i < dictionaryItemList.Count; i++)
{
       string itemNo = dictionaryItemList.ElementAt(i).itemNo.ToString();
       string description =
dictionaryItemList.ElementAt(i).description.ToString();
}
```

12.26 Check if all devices are running the same firmware

```
// Iterate through all devices and check the software revision. Note that
the gAudioServer.isSystemRevisionConsistent()
// can also be used subject to whether this check has been enabled via the
Communication Exchange web interface.
netspire.DeviceStateArray devStateArray = gAudioServer.getDeviceStates();
for (short i = 0; i < devStateArray.Count; i++)
{
    string softwareRevision =
devStateArray.ElementAt(i).getSoftwareRevision().ToString();
}
```

12.27 Check the state of a device’s digital input

```
// Following example shows how to set the digital input 3 of a device named
‘NAC01’.
// Device is identified by iterating through list returned by
‘getDeviceStates()’
string deviceId = "NAC01";  // list of deviceId can be retrieved using
function getDeviceStates()
int inputNo = 3;                   // digital input that we are interested in
bool foundDevice = false;
// Iterate though all available devices. Find the correct device
// and call the function and get the state of the digital input.
netspire.DeviceStateArray tmpDevices = gAudioServer.getDeviceStates();
for (int i = 0; i < tmpDevices.Count; i++)
{
       if (tmpDevices.ElementAt(i).getName() == deviceId)
        {
               bool result = tmpDevices.ElementAt(i).getInputState(inputNo);
               Console.WriteLine(deviceId + " Digital Input Port >" + inputNo
+ "< is >" + (result ? "set" : "not-set") + "<"));
               foundDevice = true;
               break;
        }
}
```

12.28 Set the digital output state of a device

```
// Following example shows how to set the digital output 3 of a device named
‘NAC01’.
// Device is identified by iterating through list returned by
‘getDeviceStates()’
```

```
  // list of deviceId can be retrieved using function getDeviceStates()
string deviceId = "NAC01";

// digital output to be controlled
// indicates if the output is to be set high or low. High = 1, Low = 0
int outputNo = 3;
int setBit = 1;

bool foundDevice = false;

// Iterate though all available devices. Find the correct device
// and call the function to set the state of the digital output.

netspire.DeviceStateArray tmpDevices = gAudioServer.getDeviceStates();
for (int i = 0; i < tmpDevices.Count; i++)
{
        if (tmpDevices.ElementAt(i).getName() == deviceId)
         {
                tmpDevices.ElementAt(i).setOutputState(outputNo), setBit);
                foundDevice = true;
                break;
         }
}
```

13 Appendix 1 - Dynamic Global Configuration Variables

Dynamic Global configuration variables allow an application to modify the default behaviour of the system. The dynamic configuration variables defined in this section can be passed as key value pair arguments to the AudioServer::connect method. Any dynamic configuration variable specified by the application overwrites the system default on a global basis. The provision of these variables enables an external interfacing system to manage the values of listed variables on a system wide basis.

The system does not store application specified values for Dynamic Configuration Variables in persistent storage and will revert to the system defaults on power-up.

Variable Name	Description	Default Value
PEI_Level2_Escalation_Ms	The duration in milliseconds before an Alerting HP call will be escalated from Level 1 to Level 2 Operators.	15,000
PEI_Level2_Escalation_Alert_Ms	The duration in milliseconds before an HP Escalation Alerting Tone will be played on the monitor speaker.	30,000
PEI_Level3_Escalation_Ms	The duration in milliseconds before an Alerting HP call is escalated to Level 3 operator in the absence of Level 1,2 activity	90,000
PEI_Level3_Escalation_Timeout	The duration in milliseconds before an escalated HP call offered to Level 3 Operator but not connected are removed and placed in the queue to be reconnected at a later time.	180,000
PEI_Long_Held_Button_Ms	The duration in milliseconds that a /HP button can be held before a long held button event is generated.	1,000
PEI_Type2_ Call_Activation_Delay_Ms	The duration in milliseconds that a PEI will delay prior to initaiting a Type 2 call.	0
DVA_Post_Announcement_Delay_Ms	The duration in milliseconds that the DVA will delay subsequent to a PA announcement completing.	2,000

Table 58 - Dynamic Configuration Variables

The following Dynamic Configuration Variables are still supported but obsoleted for new systems:

- PEI_Train_Radio_Escalation_Timeout
- Default_Amplifier_Gain
- EDRCallActivationDelay
- ENABLE_PEI_MONITORING
- Default_Standard_PEI_Volume
- Default_Monitor_Speaker_Gain
- Default_Handset_Speaker_Volume
- Default_Handset_Microphone_Volume
- Default_Disabled_PEI_Volume
- Default_Standard_PEI_Volume_dB
- Default_Standard_PEI_Spk_Volume_dB
- PEI_Driver_Escalation_Ms
- Escalation_Alert_Ms
- PEI_Train_Radio_Escalation_Ms
- PEI_Train_Radio_Escalation_Timeout_Ms
- Default_Amplifier_Gain_dB
- Commercial_Radio_Interrupt_Ms
- Enable_PEI_Monitoring
- PA_Sinks

14 Appendix 2 - Dictionary

14.1 Dictionary Changeset XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    targetNamespace="http://www.oa.com.au/asapi"
    xmlns:asapi="http://www.oa.com.au/asapi">
    <xs:element name="dictChangeSet">
        <xs:complexType>
            <xs:sequence>
                <xs:element minOccurs="1" maxOccurs="unbounded"
ref="asapi:change"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="change">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="versionSeq" type="xs:integer"/>
                <xs:element name="operation">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="AddDictionaryItem"/> <!--
Instructs the system to automatically allocate a free number in the range
100,000 to 200,000 -->
                        <xs:enumeration value="UpdateDictionaryItem"/>
                        <xs:enumeration value="DeleteDictionaryItem"/>
                        <xs:enumeration value="ClearDictionary"/>
                        <xs:enumeration value="ResetDictionary"/>
                    </xs:restriction>
                </xs:simpleType>
                <xs:element minOccurs="0" maxOccurs="1" name="newVersion">
                    <xs:simpleType>
                        <xs:restriction base="xs:integer">
                            <xs:minInclusive value="1"/>
                            <xs:maxInclusive value="2147483647"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element minOccurs="0" maxOccurs="1" name="dictItemNo">
                    <xs:simpleType>
```

```
                        <xs:restriction base="xs:integer">
                            <xs:minInclusive value="1"/>
                            <xs:maxInclusive value="2147483647"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element minOccurs="0" maxOccurs="1"
ref="asapi:dictItem"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="dictItem">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="deviceType" default="TGU">
                    <!-- deviceType field is now obsolete -->
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="PEI"/>
                            <xs:enumeration value="TGU"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="description" minOccurs="0">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:minLength value="1"/>
                            <xs:maxLength value="512"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="category">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:minLength value="1"/>
                            <xs:maxLength value="256"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element minOccurs="0" maxOccurs="3"
ref="asapi:audioSegment"/>
                <xs:element minOccurs="0" maxOccurs="1" ref="asapi:image"/>
                <xs:element minOccurs="0" maxOccurs="1" ref="asapi:video"/>
                <xs:element minOccurs="0" maxOccurs="1"
name="asapi:displayText"/>
```

```
                <xs:element name="metadata" minOccurs="0" maxOccurs="1">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:minLength value="0"/>
                            <xs:maxLength value="512"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="audioSegment">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="text">
                    <!-- This field is now obsolete -->
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:minLength value="0"/>
                            <xs:maxLength value="256"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="format" default="48k-PCM">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="48k-PCM"/>
                            <xs:enumeration value="8k-ALAW"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="inflection" default="flat">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="flat"/>
                            <xs:enumeration value="falling"/>
                            <xs:enumeration value="rising"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="fileName">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:minLength value="1"/>
                            <xs:maxLength value="256"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="duration" type="xs:nonNegativeInteger"
default="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="image">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="imageType">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="generic"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="imageFileName">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:minLength value="1"/>
                            <xs:maxLength value="256"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="video">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="videoType">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="generic"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="videoFileName">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:minLength value="1"/>
                            <xs:maxLength value="256"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="videoDuration"
type="xs:nonNegativeInteger" default="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="displayText">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:minLength value="1"/>
                <xs:maxLength value="512"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
</xs:schema>
```

14.2 Sample Dictionary Changeset XML Document

An example changeset definition file is given below:

```
<?xml  version="1.0"  encoding="UTF-8"?>

<dictChangeSet
    xmlns=http://www.oa.com.au/asapi
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="http://www.oa.com.au/asapi changeset.xsd" >

<change>
    <!--Delete earlier dictionary item 999 -->
    <versionSeq>14</versionSeq>
    <operation>DeleteDictionaryItem</operation>
    <dictItemNo>999</dictItemNo>

</change>

<change>
    <!-- Clear all dictionary items and re-assign current version number --
>
    <versionSeq>15</versionSeq>
    <operation>ClearDictionary</operation>
    <newVersion>200</newVersion>
```

```
</change>

<change>
    <!-- Add dictionary item 1001 -->
    <versionSeq>201</versionSeq>
    <operation>UpdateDictionaryItem</operation>
    <dictItemNo>1001</dictItemNo>
    <dictItem>
        <deviceType>TGU</deviceType>
        <category>Warning messages</category>
        <audioSegment>
            <text>Doors closing, please stand clear</text>
            <inflection>flat</inflection>
            <fileName>doors_closing.wav</fileName>
        </audioSegment>
        <displayText>Stand clear of doors</displayText>
    </dictItem>
</change>

<change>
    <!-- Add dictionary item 1002 -->
    <versionSeq>202</versionSeq>
    <operation>UpdateDictionaryItem</operation>
    <dictItemNo>1002</dictItemNo>
    <dictItem>
        <deviceType>TGU</deviceType>
        <category>Warning messages</category>
        <audioSegment>
            <text>This train is now due to depart, please stand
clear</text>
            <inflection>flat</inflection>
            <fileName>due_to_depart.wav</fileName>
        </audioSegment>
    </dictItem>
</change>

</dictChangeSet>
```

15 Appendix 3 – Porting Guidelines

This section provides an overview of porting an application developed using Revision 1.0 to 1.13 of NetSpire SDK.

15.1 Initial Connection

Applications can continue to call the method AudioServer::connect() to initiate connection to one or more Communication Exchanges.

The set of key value pairs sent to the connect method using the KeyValueMap argument now accepts the arguments listed in Table 53 - SDK Configuration Item Keys.

Creation of a configuration file and sending its path using the key ASAPI_CONFIGURE_FILE is no longer required, or supported. The configuration file was used to set the port number used by the SDK when communicating with Communication Exchanges. This can now be achieved by using the key NETSPIRE_SDK_SOCKET_PORT.

Dynamic configuration variables listed in section Appendix 1 - Dynamic Global Configuration Variables are no longer accepted in the KeyValueMap argument of AudioServer::connect(). Dynamic Configuration variables can now be set using the SDK method AudioServer::setDynamicConfiguration defined in section 8.2.4.5.

15.2 Dynamic Configuration Variables

The list of Dynamic Configuration Variables supported in the previous SDK have been listed below. Where the variable is renamed or obsoleted, new method to achieve the same functionality are also listed.

Dynamic Configuration Variable	Description	Updated Method
Default_Monitor_Speaker_Gain	The default (static) gain for Ceiling Speakers. Valid values are [+18,-78] dB.	This method to set default gain for monitor speakers is supported. A more flexible way to control gain levels is also available in the current SDK and its use is recommended to simplify gain control. The methods PaSink::getGain() and PaSink::setGain() defined in sections 9.4.7.1 and 9.4.7.2 can be used to set the gain level of any audio output devices including monitor speakers. This method allows setting individual speaker outputs to different gain levels.
Default_Handset_Speaker_Volume	The default value for the Crew Intercom handset speaker volume. Valid values are [1-10].	This Dynamic Configuration Variable is supported and can be used.

Dynamic Configuration Variable	Description	Updated Method
Default_Handset_Microphone_Volume	The default value for the Crew intercom microphone speaker volume. Valid values are [1-10].	This Dynamic Configuration Variable is supported and can be used.
Default_Disabled_PEI_Volume	The default volume level of Disabled PEI speakers. Valid values are [1, 8]; where 1 is the minimum volume level and 8 is the maximum volume level.	This Dynamic Configuration Variable is still supported and can be used.
Default_Standard_PEI_Volume	The default volume level of Standard PEI speakers. Valid values are [1, 8]; where 1 is the minimum volume level and 8 is the maximum volume level.	This Dynamic Configuration Variable is still supported and can be used.
PEI_Driver_Escalation_Ms	The duration in milliseconds before an Alerting PEI call will be escalated from guard to driver Crew Intercom.	This Dynamic Configuration Variable is still supported and can be used. A new alternate name for this function is PEI_Level2_Escalation_Ms, which can also be used.
Escalation_Alert_Ms	The duration in milliseconds before an PEI Escalation Alerting Tone will be played on the Crew Speaker.	This Dynamic Configuration Variable is still supported and can be used. A new alternate name for this function is PEI_Level2_Escalation_Alert_Ms, which can also be used.
PEI_Train_Radio_Escalation_Ms	The duration in milliseconds before an Alerting PEI call is escalated to Train Radio in the absence of crew activity	This Dynamic Configuration Variable is still supported and can be used. A new alternate name for this function is PEI_Level3_Escalation_Ms, which can also be used.
PEI_Train_Radio_Escalation_Timeout	The duration in milliseconds before an escalated PEI call offered to the Train Radio but not connected are removed and placed in the queue to be reconnected at a later time.	This Dynamic Configuration Variable is still supported and can be used. A new alternate name for this function is PEI_Level3_Escalation_Timeout, which can also be used.

Dynamic Configuration Variable	Description	Updated Method
PEI_Long_Held_Button_Ms	The duration in milliseconds that a PEI button can be held before a long held button event is generated.	This Dynamic Configuration Variable is still supported and can be used.
DVA_Post_Announcement_Delay	The duration in milliseconds that the DVA will delay subsequent to a PA announcement completing.	This Dynamic Configuration Variable is still supported and can be used.
Default_Amplifier_Gain	The default (static) gain for all amplifiers in the TGU. TGU amplifier gains are not individually set. A baseline gain is set for all amplifiers and this value will determine the relative increase or decrease. Valid range is [+18,-78] dB.	This Dynamic Configuration Variable is still supported and can be used. A more flexible way to control gain levels is also available in the current SDK and its use is recommended to simplify gain control. The methods PaSink::getGain() and PaSink::setGain() defined in sections 9.4.7.1 and 9.4.7.2 can be used to set the gain level of any audio output devices including monitor speakers. This method allows setting individual speaker outputs to different gain levels.
Commercial_Radio_Interrupt_Ms	The automatic interrupt duration in milliseconds that will be used when the commercial radio is muted.	This Dynamic Configuration Variable is still supported and can be used.
EDRCallActivationDelay	The delay in milliseconds that once a EDR stage 1 event happens, an EDR call will be made after this delay.	This Dynamic Configuration Variable is still supported and can be used. A new alternate name for this function is PEI_Type2_ Call_Activation_Delay_Ms, which can also be used.

15.3 Retrieving Device List

Calling the method AudioServer::uploadDeviceList to upload an association of device names and their IP addresses is no longer required or supported.

Client applications can instead retrieve a list of all currently connected device names and their IP addresses, similar to the code segment provided below:

```
        DeviceStateArray states = as.getDeviceStates();
        for (int i = 0; i < states.size(); i++)
        {
             Device device = states.get(i);
             System.out.println("Name: " + device.getName() + " State: " + device.getState() + " IP
Address: " + device.getIP() );
        }
        states.delete();
```

15.4 Updates to API Methods

The following functions have been replaced with functions that invoke an equivalent behaviour.

a) AudioServer::setCommercialRadioEnabled
Calls to AudioServer::setCommercialRadioEnabled(string devicename, bool on/off) needs to be replaced with Device::setCondition(string conditionName, bool value)where,

The device object is of type Device, representing the MCX device connected to a commercial radio,

conditionName is set to “CommercialRadioEnabled”,

value is set to true to enable and false to disable the radio.

TCX and MCX devices are configured to take action using its configurable ConditionMonitor rules, and will enable or disable the connected commercial radio.

b) AudioServer::setCommercialRadioInterrupt()

Calls to AudioServer:: setCommercialRadioInterrupt(string devicename, bool initiate) needs to be replaced with Device::setCondition(string conditionName, bool value) where,

The device object is of type Device, representing the TCX device connected to a commercial radio,

conditionName is set to “CommercialRadioSoftInterrupt”,

value is set to true to enable and false to disable the interrupt.

TCX devices are configured to take action using its configurable ConditionMonitor rules, and will enable or disable the connected commercial radio interrupt timer based on this setting.

c) AudioServer::setCrewSpeakerDVAPAMonitoring()
Enables / disables DVA and PA monitoring in Crew Cabins. Does not impact PEI / Crew intercom monitoring.

This method can be replaced with PaSink::setPAMonitoring().

d) AudioServer::getCrewSpeakerDVAPAMonitoring()
Returns current state of DVA and PA monitoring in Crew Cabins.

This method can be replaced with PaSink::getPAMonitoring().

e) AudioServer::setDoorOpen()
Advises which side the doors will be opening. This is used to set the channel assignment for "EXT PA".

This method can be replaced by calling Device::setCondition(“DoorOpenSide1”, true) and Device::setCondition(“DoorOpenSide2”, true) on the active TCX.

f) AudioServer::setDoorState()

Advises which side the doors are opened or closed. This is used to set the channel assignment for "EXT PA" and control the latching behaviour of "PA" selection buttons. This method can be replaced by the following two API calls:

- Invoke Device::setCondition(“DoorOpenSide1”, true) and Device::setCondition(“DoorOpenSide2”, true) on the active TCX.
- Behaviour of PA selection button latching behaviour can be set to latching by calling Device::setCondition(“PaButtonLatch”, true) on the active TCX. Behaviour of PA selection button latching behaviour can be set to non-latching by calling Device::setCondition(“PaButtonLatch”, false) on the active TCX.

g) AudioServer::setCrewSpeakerGain() and AudioServer::getCrewSpeakerGain()

The methods PaSink::getGain() and PaSink::setGain() defined in sections 9.4.7.1 and 9.4.7.2 can be used to set the gain level of any audio output devices including crew speakers. This method allows setting individual speaker outputs to different gain levels.

h) AudioServer::getCommercialRadioInterrupt()
This method returns the value of the TCX digital output connected to an external device. This method can be replaced with Device::getOutputState(int outputNumber), where

The device object is of type Device, representing the TCX device connected to a commercial radio, and

outputNumber is the DIO number of the corresponding TCX output. The commercial radio interrupt is connected to output 1 of TCX devices.

i) AudioServer::isSlowSpeed ()
This method returns the value of the TCX digital input that indicates if the rolling stock is moving at less than 5km/h. connected to an external device. This method can be replaced with Device::getInputState(int inputNumber), where

The device object is of type Device, representing the TCX device that is connected to the slow speed indication, and

inputNumber is the DIO number of the corresponding TCX input. The slow speed input is connected to input 5 of the TCX.

j) AudioServer::getInputState (const std::string & deviceId, InputNo inputNo)
This method has been moved to Device::getInputState (InputNo inputNo).

k) AudioServer::trainCertificationTest() and AudioServer::getTCTState()

Train Certification Test functionality invokes device self test for all devices. This method can be replaced by calling Device::initiateDeviceHealthTest() for all devices, or calling AudioServer::initiateDeviceHealthTest with an empty argument.

Test status of each device is available using Device::getHealthTestStatus().

l) AudioServer::playMessage()
Arguments to this method has been updated, and a list of PaZones are now required when calling the method. List of PaZones can be determined by calling PAController::getPaZones() to retrieve the list of currently defined zones.

The recommended and easier to use replacement for this method is to use PAController::playMessage which accepts a list of zones as arguments. Please refer to section 9.2.4.23 for details on this method.

15.5 Dictionary Update

Dictionary update process has been updated to decrease the time where DVA announcements cannot be made during the update. A new state (DICT_UPDATE_PENDING) has been added to the enumerated type DictionaryUpdateStatus, which can be used by client applications to determine dictionary update has started but the DVA subsystem can still be utilised with the existing revision of the dictionary.

States included in the enumerated type DictionaryUpdateStatus are now defined as follows:

- DICT_UPDATE_NONE No dictionary updates initiated or the last update is completed. Please query the dictionary version using getDictionaryVersion() to verify the expected dictionary version is loaded on the device. Currently reported dictionary version can continue to be used in announcement requests.
- DICT_UPDATE_IN_PROGRESS Last dictionary update is in progress. Media files are being replaced in this state and behaviour of announcement requests while the device is in this state is undefined.
- DICT_UPDATE_FAILED Last dictionary update failed. Currently reported dictionary version can continue to be used in announcement requests.
- DICT_UPDATE_PENDING Dictionary update is pending and is being processed in the background. Currently reported dictionary version can continue to be used in announcement requests.

There has been no change to the method Device::getDictionaryUpdateStatus(), which can be used to query the update status of each device and to the method Device:getDictionaryVersion(), which is used to query the dictionary revision of each device.

With the updated API, the following dictionary update workflow can be used in order to limit the time when announcements cannot be made:

- Invoke updateDictionary() to update the revision of dictionary in the system.

- Call Device::getDictionaryUpdateStatus() on the active TCX/MCX and verify it is now reporting DICT_UPDATE_PENDING

- Call Device::getDictionaryUpdateStatus() on all devices, and continue to make announcements using the old dictionary as long as all devices report a state other than DICT_UPDATE_IN_PROGRESS and report the old revision.

- Do not initiate any new announcements using either the old or the new dictionary when any of the devices report DICT_UPDATE_IN_PROGRESS and report the new revision.

- Start making announcements using the new dictionary when all devices report a state other than DICT_UPDATE_IN_PROGRESS and report the new revision.

16 Appendix 3 – Sample Programs

16.1 Sample Program - Monitor All Device States – C++

```
// NetSpire SDK Sample Program.
// Name:      Monitor All Device Stations.
// Description: Provides event driven and polling methods for iterating through 
all
//            available NetSpire devices in a system and displaying device states
//            as they change.
// Language   C++

#include <iostream>
#include <string>

#include "AudioServer.hpp"

// Global Variables
AudioServer gAudioServer;
bool gUsePolling = true;

/*******************************************************************************
*****************/
class asObserverClass : public AudioServerObserver
{
    public:
        asObserverClass()
        {
            //std::cout << "asObserverClass::asObserverClass()" << std::endl;
        }

        void onCommsLinkUp()
        {
            std::cout << "asObserverClass::onCommsLinkUp()" << std::endl;
        }

        void onCommsLinkDown()
        {
            std::cout << "asObserverClass::onCommsLinkDown()" << std::endl;
        }

        void onAudioConnected()
        {
            std::cout << "asObserverClass::onAudioConnected()" << std::endl;
        }

        void onAudioDisconnected()
        {
            std::cout << "asObserverClass::onAudioDisconnected()" << std::endl;
        }

        void onDeviceStateChange(Device device)
        {
      std::cout << "asObserverClass::onDeviceStateChange(): " << "deviceId >" << 
      device.getName() << "< deviceState >" << device.getStateText() << "<" << 
      std::endl;
        }
        void onDeviceDelete(Device device)
        {
       std::cout << "asObserverClass::onDeviceDelete(): deviceId >" <<
       device.getName() << "<" << std::endl;
        }

        void onVUMeterUpdate(int level)
        {
            std::cout << "asObserverClass::onVUMeterUpdate()" << std::endl;
        }

        void onConfigUpdate(std::string progress, std::string key, std::string
value)
        {
       std::cout << "asObserverClass::onConfigUpdate(): progress >" << progress <<
       "< key >" << key << "< value >" << value << "<" << std::endl;
        }

        void onUserDataUpdate(int elemId, int key, std::string value)
        {
       std::cout << "asObserverClass::onUserDataUpdate(): elemId >" << elemId <<
       "< key >" << key << "< value >" << value << "<" << std::endl;
        }
};

/*********************************************************************************
******************/
class paObserverClass : public PAControllerObserver
{
    public:
        paObserverClass()
        {
            //std::cout << "paObserverClass::paObserverClass()" << std::endl;
        }

        void onPaSourceUpdate(PaSource source)
        {
       std::cout << "paObserverClass::onPaSourceUpdate(): PaSourceId >" <<
       source.id << "<" << std::endl;
        }

        void onPaSourceDelete(int sourceId)
        {
       std::cout << "paObserverClass::onPaSourceDelete(): PaSourceId >" <<
       sourceId << "<" << std::endl;
        }

        void onPaSinkUpdate(PaSink sink)
        {
       std::cout << "paObserverClass::onPaSinkUpdate(): PaSinkId >" << sink.id <<
       "<" << std::endl;
        }

        void onPaSinkDelete(int sinkId)
        {
       std::cout << "paObserverClass::onPaSinkDelete(): PaSinkId >" << sinkId <<
       "<" << std::endl;
        }
        void onPaTriggerUpdate(PaTrigger trigger)
        {
       std::cout << "paObserverClass::onPaTriggerUpdate(): PaTriggerId >" <<
       trigger.id << "<" << std::endl;
        }

        void onPaTriggerDelete(int triggerId)
        {
       std::cout << "paObserverClass::onPaTriggerDelete(): PaTriggerId >" <<
       triggerId << "<" << std::endl;
        }

        void onPaSelectorUpdate(PaSelector selector)
        {
       std::cout << "paObserverClass::onPaSelectorUpdate(): PaSelectorId >" <<
       selector.id << "<" << std::endl;
        }

        void onPaSelectorDelete(int selectorId)
        {
       std::cout << "paObserverClass::onPaSelectorDelete(): PaSelectorId >" <<
       selectorId << "<" << std::endl;
        }
};

/*******************************************************************************
*****************/
class callObserverClass : public CallControllerObserver
{
    public:
        void onCallUpdate(CallInfo callInfo)
        {
       std::cout << "callObserverClass::onCallUpdate(): callId >" <<
       callInfo.callId << "<" << std::endl;
        }

        void onCallDelete(int callId)
        {
            std::cout << "callObserverClass::onCallDelete(): callId >" << callId
<< "<" << std::endl;
        }

        void onCDRMessageUpdate(CDRInfo cdrInfo)
        {
       std::cout << "callObserverClass::onCDRMessageUpdate(): cdrId >" <<
       cdrInfo.id << "," << std::endl;
        }

        void onCDRMessageDelete(int cdrId)
        {
       std::cout << "callObserverClass::onCDRMessageDelete(): cdrId >" << cdrId <<
       "<" << std::endl;
        }

        void onTerminalUpdate(TerminalInfo termInfo)
        {
       std::cout << "callObserverClass::onTerminalUpdate(): TermId >" <<
       termInfo.termId << "<" << std::endl;
```

```
        }

        void onTerminalDelete(int termId)
        {
       std::cout << "callObserverClass::onTerminalDelete(): TermId >" << termId <<
       "<" << std::endl;
        }

        void onSIPTrunkUpdate(SIPTrunk sipTrunk)
        {
       std::cout << "callObserverClass::onSIPTrunkUpdate(): trunkName >" <<
       sipTrunk.getName() << "< updated in deviceId >" << sipTrunk.getDeviceId()
       << "<" << std::endl;
        }

        void onSIPTrunkDelete(SIPTrunk sipTrunk)
        {
       std::cout << "callObserverClass::onSIPTrunkDelete(): trunkName >" <<
       sipTrunk.getName() << "< deleted from deviceId >" << sipTrunk.getDeviceId()
       << "<" << std::endl;
        }

        void onISDNTrunkUpdate(ISDNTrunk isdnTrunk)
        {
       std::cout << "callObserverClass::onISDNTrunkUpdate(): trunkName >" <<
       isdnTrunk.getName() << "< updated in deviceId >" << isdnTrunk.getDeviceId()
       << "<" << std::endl;
        }

        void onISDNTrunkDelete(ISDNTrunk isdnTrunk)
        {
       std::cout << "callObserverClass::onISDNTrunkDelete(): trunkName >" <<
       isdnTrunk.getName() << "< deleted from deviceId >" <<
       isdnTrunk.getDeviceId() << "<" << std::endl;
        }
};


/*********************************************************************************
*****************/
int main (int argc, char *argv[])
{
       // setup connection parameters
       AudioServer::KeyValueMap connectionParametersKeyValuePair;

       // change API binding port number (default port number is 20770)
       connectionParametersKeyValuePair.insert(
       connectionParametersKeyValuePair.end(), std::pair< std::string, std::string
       > ("NETSPIRE_SDK_SOCKET_PORT", "20772") );

       // disabled debug logging
       connectionParametersKeyValuePair.insert(
       connectionParametersKeyValuePair.end(), std::pair< std::string, std::string
       > ("NETSPIRE_SDK_SET_LOG_LEVEL", "0") );

       // Set addresses for 2 CXS servers operating redundantly.
       AudioServer::StringArray audioserverAddresses;
       audioserverAddresses.push_back("10.20.1.121");
       audioserverAddresses.push_back("10.20.1.123");
```

```
        // Check to see if asynchronous event based processing is being used.
        // If it is, then register event observers.

        if (!gUsePolling)
        {
                asObserverClass asObserver;
                paObserverClass paObserver;
                callObserverClass callObserver;
                gAudioServer.registerObserver(&asObserver);
                gAudioServer.getPAController()->registerObserver(&paObserver);
                gAudioServer.getCallController()->registerObserver(&callObserver);
        }

        // connect to Communication Exchange
        gAudioServer.connect (audioserverAddresses,
        connectionParametersKeyValuePair);

        if (gUsePolling)
        {
                // wait until connection is established on the communication
                exchange.
                for (unsigned int i = 0; i < 10; i++)
                {
                        if (gAudioServer.isAudioConnected())
                        {
                            break;
                        }
                        Sleep(60);
                }

                if (!gAudioServer.isAudioConnected())
                {
                      std::cerr << "Cannot connect to the Server. " << std::endl;
                      return EXIT_FAILURE;
                }
        }

//==================================================================================
===============
// connection to Communication Exchange is established. idle forever and get
device states every x seconds
//==================================================================================
===============
        for (;;)
        {
            if (gUsePolling)
            {
                AudioServer::DeviceStateArray devState =
        gAudioServer.getDeviceStates();
                for (std::vector<Device>::iterator i = devState.begin(); i !=
        devState.end(); i++)
                {
                    Device tmp = (*i);
                    std::ostringstream oss;
                    oss << tmp.getName();
                    oss << "/";
                    oss << tmp.getNameNotModified();
                    oss << " dstNo: " << tmp.getDstNo();
                    oss << " locName: " << tmp.getLocationName();
                    oss << " locId: " << tmp.getLocationId();
                    oss << " IP: " << tmp.getIP();
                    oss << " dictSupported: " << tmp.getDictionarySupport();
                    oss << " dictVersion: " << tmp.getDictionaryVersion();
                    oss << " dictStatus: " << tmp.getDictionaryUpdateStatus();
                    oss << " state: " << tmp.getStateText();
                    oss << " softwareRevision: " << tmp.getSoftwareRevision();
                    oss << " devClass: " << tmp.getDeviceClass();
                    oss << " devModelName: " << tmp.getDeviceModel().getName();
                    std::cout << oss.str() << std::endl;
                }
                std::cout << std::endl;
            }
            sleepMS(10000);
        }

        gAudioServer.disconnect ();
        return EXIT_SUCCESS;
}
```

16.2 Sample Program - Monitor All Device States – Java

```java
// NetSpire SDK Sample Program.
// Name:      Monitor All Device Stations.
// Description: Provides event driven and polling methods for iterating through 
all
//            available NetSpire devices in a system and displaying device states
//            as they change.
// Language   Java

import oa.as.AudioServer;
import oa.as.AudioServerObserver;
import oa.as.PAController;
import oa.as.PAControllerObserver;
import oa.as.CallControllerObserver;
import oa.as.CDRInfo;
import oa.as.CallInfo;
import oa.as.Device;
import oa.as.DeviceStateArray;
import oa.as.Gain;
import oa.as.KeyValueMap;
import oa.as.NumberArray;
import oa.as.PaSelector;
import oa.as.PaSink;
import oa.as.PaSinkArray;
import oa.as.PaSource;
import oa.as.PaTrigger;
import oa.as.StringArray;
import oa.as.TerminalInfo;
import oa.as.NamedId;
import oa.as.Vehicle;
import oa.as.Station;
import oa.as. PlatformInfo;
import oa.as.Service;

class main
{
    // Global to indicate if program will use polling or event-driven
    public static boolean gUsePolling = true;

    /***************************************************************************/
    private static class audioserverObserver extends AudioServerObserver
    {
        @Override
        public void onCommsLinkUp()
        {
            System.out.println("main::onCommsLinkUp()");
        }

        @Override
        public void onCommsLinkDown()
        {
            System.out.println("main::onCommsLinkDown()");
        }

        @Override
        public void onAudioConnected()
        {
            System.out.println("main::onAudioConnected()");
        }

        @Override
        public void onAudioDisconnected()
        {
            System.out.println("main::onAudioDisconnected()");
        }

        @Override
        public void onDeviceStateChange(Device device)
        {
            System.out.println("main::onDeviceStateChange() - State of " +
device.getName() + " is now " + device.getState());
        }

        @Override
        public void onDeviceDelete(Device device)
        {
            System.out.println("main::onDeviceDelete() - " + device.getName());
        }
    }

    /*************************************************************************/
    private static class paControllerObserver extends PAControllerObserver
    {
        @Override
        public void onPaSourceUpdate(final PaSource nativeSourceObject)
        {
            System.out.println("main::onPaSourceUpdate()");
            PaSource source = new PaSource(nativeSourceObject);
            int sourceId = source.getId();
            //System.out.println("Received PA sink update: " + sinkId + " (sink
hash = " + sink.hashCode() + ")");
        }

        @Override
        public void onPaSourceDelete(int sourceId)
        {
            System.out.println("main::onPaSourceDelete()");
        }

        @Override
        public void onPaSinkUpdate(final PaSink nativeSinkObject)
        {
            System.out.println("main::onPaSinkUpdate()");
            PaSink sink = new PaSink(nativeSinkObject);
            int sinkId = sink.getId();
            //System.out.println("Received PA sink update: " + sinkId + " (sink
hash = " + sink.hashCode() + ")");
        }

        @Override
        public void onPaSinkDelete(int sinkId)
        {
            System.out.println("main::onPaSinkDelete()");
        }

        @Override
        public void onPaTriggerUpdate(final PaTrigger nativeTriggerObject)
        {
            System.out.println("main::onPaTriggerUpdate()");
        }

        @Override
        public void onPaTriggerDelete(int triggerId)
        {
            System.out.println("main::onPaTriggerDelete()");
        }

        @Override
        public void onPaSelectorUpdate(final PaSelector nativeSelectorObject)
        {
            System.out.println("main::onPaSelectorUpdate()");
        }

        @Override
        public void onPaSelectorDelete(int selectorId)
        {
            System.out.println("main::onPaSelectorDelete()");
        }
    }

    /***************************************************************************/
    private static class callControllerObserver extends CallControllerObserver
    {
        @Override
        public void onCallUpdate(final CallInfo callInfo)
        {
            System.out.println("main::onCallUpdate()");
        }

        @Override
        public void onCallDelete(int callId)
        {
            System.out.println("main::onCallDelete()");
        }

        @Override
        public void onCDRMessageUpdate(final CDRInfo cdrInfo)
        {
            System.out.println("main::onCDRMessageUpdate()");
        }

        @Override
        public void onCDRMessageDelete(int cdrId)
        {
            System.out.println("main::onCDRMessageDelete()");
        }

        @Override
        public void onTerminalUpdate(final TerminalInfo termInfo)
        {
            System.out.println("main::onTerminalUpdate()");
        }

        @Override
        public void onTerminalDelete(int termId)
        {
            System.out.println("main::onTerminalDelete()");
        }
    }

    /*****************************************************************************/
    public static void main (String argv[])
    {
        System.loadLibrary("FTPlib");
        System.loadLibrary("netspireSDK");
        System.out.println("Testing AudioServer class methods");

        // setup connection parameters
        KeyValueMap connectionParametersKeyValuePair = new KeyValueMap();

       // change API binding port number (default port number is 20770)
        connectionParametersKeyValuePair.set("NETSPIRE_SDK_SOCKET_PORT", "20772");
        connectionParametersKeyValuePair.set("NETSPIRE_SDK_SET_LOG_LEVEL", "0");

        StringArray serverAddresses = new StringArray();
        serverAddresses.add("10.20.1.121");
        serverAddresses.add("10.20.1.123");

        // register observers
        AudioServer as = new AudioServer();

      // Check to see if asynchronous event based processing is being used.
      // If it is, then register event observers.
        if (!gUsePolling)
        {
            audioserverObserver asObserver = new audioserverObserver();
            paControllerObserver paObserver = new paControllerObserver();
            callControllerObserver callObserver = new callControllerObserver();

            as.registerObserver(asObserver);
            as.getPAController().registerObserver(paObserver);
            as.getCallController().registerObserver(callObserver);
        }


      // Connect to Communication Exchange.
        as.connect(serverAddresses, connectionParametersKeyValuePair);
        serverAddresses.delete();
        connectionParametersKeyValuePair.delete();

        if (gUsePolling)
        {
            // wait until connection is established on the communication exchange.
            for (int i = 0; i < 10; i++)
            {
                // check if connection is established
                if (as.isAudioConnected())
                {
                    break;
                }

                // sleep and then check again is connection was success
                try { Thread.sleep(60); }
                catch (InterruptedException e) { }
            }

            if (!as.isAudioConnected())
            {
                System.err.println("Cannot connect to the Communication
Exchange.");
                return;
            }
        }


//=================================================================================
===============
// connection to Communication Exchange is established. idle forever and get
device states every 2 seconds
//=================================================================================
===============
        for (;;)
        {
            if (gUsePolling)
            {
                DeviceStateArray states = as.getDeviceStates();
                for (int i = 0; i < states.size(); i++)
                {
                    Device device = states.get(i);
                    System.out.println("Name: " + device.getName() + " State: " +
device.getState());
                }
                states.delete();
            }

            try { Thread.sleep(10000); }
            catch (InterruptedException e) { }

        }

    }
}
```

16.3 Sample Program - Monitor All Device States – C#

```
// NetSpire SDK Sample Program.
// Name:      Monitor All Device Stations.
// Description: Provides event driven and polling methods for iterating through
all
//            available NetSpire devices in a system and displaying device states
//            as they change.
// Language   C#

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO;

namespace netspireSDKTester
{
    class Program
    {
        // globals
        public static bool gUsePolling = false;
        public static netspire.AudioServer gAudioServer = new
netspire.AudioServer();
        public static netspire.AudioServerObserver gAudioServerObserver = null;
        public static netspire.PAControllerObserver gPAControllerObserver = null;
        public static netspire.CallControllerObserver gCallControllerObserver =
null;

        static void Main(string[] args)
        {
            // set communication server addresses
            netspire.StringArray serverAddresses = new netspire.StringArray();
            serverAddresses.Add("10.20.1.121");
            serverAddresses.Add("10.20.1.123");

            // set logging if enabled
            netspire.KeyValueMap configItems = new netspire.KeyValueMap();
            configItems.Add("NETSPIRE_SDK_SOCKET_PORT", "20772");
            configItems.Add("NETSPIRE_SDK_SET_LOG_LEVEL", "0");

            // got all connection parameters - create a new server object
            if (gAudioServer == null)
            {
                gAudioServer = new netspire.AudioServer();
            }


        // Check to see if asynchronous event based processing is being used.
        // If it is, then register event observers.
            if (!gUsePolling)
            {
                // create a new observer for event-based processing
                if (gAudioServerObserver == null)
                {
                    gAudioServerObserver = new asObserverClass();
                    Console.WriteLine("created new observer - asObserver");
                }
                if (gPAControllerObserver == null)
                {
                    gPAControllerObserver = new paObserverClass();
                    Console.WriteLine("created new observer - paObserver");
                }
                if (gCallControllerObserver == null)
                {
                    gCallControllerObserver = new callObserverClass();
                    Console.WriteLine("created new observer - callObserver");
                }
                gAudioServer.registerObserver(gAudioServerObserver);

gAudioServer.getPAController().registerObserver(gPAControllerObserver);

gAudioServer.getCallController().registerObserver(gCallControllerObserver);
            }

            // all OK - send connect request to DVA server
            gAudioServer.connect(serverAddresses, configItems);
            if (!gUsePolling)
            {
                // nothing to do - do stuff when events are received.
            }
            else
            {
                for (short i = 0; i < 100; i++)
                {
                    if (gAudioServer.isAudioConnected())
                        break;
                    System.Threading.Thread.Sleep(500);
                }
                if (!gAudioServer.isAudioConnected())
                {
                    Console.WriteLine("connection to AS failed");
                    return;
                }
                else
                {
                    Console.WriteLine("connection Established");
                }
            }
//=================================================================================
================
// connection to Communication Exchange is established. idle forever and get
device states every 2 seconds
//=================================================================================
================
            for (;;)
            {
                if (gUsePolling)
                {
                    // poll for device status
                    netspire.DeviceStateArray devStateArray =
gAudioServer.getDeviceStates();
                    for (short i = 0; i < devStateArray.Count; i++)
                    {
                        string supplementaryFieldsStr = "";
                    netspire.KeyValueMap tmpKeyValueMap =
                    devStateArray.ElementAt(i).getSupplementaryFields();
                        for (int j = 0; j < tmpKeyValueMap.Count; j++)
                        {
                    SupplementaryFieldsStr = supplementaryFieldsStr + "[" +
                    tmpKeyValueMap.ElementAt(j).Key + "," +
                    tmpKeyValueMap.ElementAt(j).Value + "] ";
                        }
                        Console.WriteLine("Name: " +
devStateArray.ElementAt(i).getName().ToString() +
                    ", State: " +
             devStateArray.ElementAt(i).getStateText().ToString() +
                    ", Dictionary Version: " +
                    devStateArray.ElementAt(i).getDictionaryVersion().ToString()
                        );
                    }
                    Console.WriteLine("\n");
                }

                // sleep a bit
                System.Threading.Thread.Sleep(2000);
            }
        }


/*****************************************************************************
*************
   AudioServer Observer
*****************************************************************************
*************/
        public class asObserverClass : netspire.AudioServerObserver
        {
/*****************************************************************************
*************/
            public asObserverClass()
            {
            }


/*****************************************************************************
**************/
            public override void onCommsLinkUp()
            {
                Console.WriteLine("asObserverClass.onCommsLinkUp()");
            }


/*****************************************************************************
*****************/
            public override void onCommsLinkDown()
            {
                Console.WriteLine("asObserverClass.onCommsLinkDown()");
            }


/*****************************************************************************
******************/
            public override void onAudioConnected()
            {
                Console.WriteLine("asObserverClass.onAudioConnected()");
            }


/*****************************************************************************
*****************/
            public override void onAudioDisconnected()
            {
                Console.WriteLine("asObserverClass.onAudioDisconnected()");
            }


/*****************************************************************************
*****************/
            public override void onDeviceStateChange(netspire.Device device)
            {
                netspire.Device tmpDevice = new netspire.Device(device);
                netspire.KeyValueMap tmpKeyValueMap =
tmpDevice.getSupplementaryFields();
                String supplementaryFieldMsg = "";
                String msg = "asObserverClass.onDeviceStateChange() - " +
                    "Name >" + tmpDevice.getName() + "< " +
                    "State >" + tmpDevice.getState().ToString() + "< " +
                    "Dict Status >" + tmpDevice.getDictionaryUpdateStatus() + "< "
+
                    "Dict VersionSeq >" + tmpDevice.getDictionaryVersion() + "< "
+
                    "Health Test Status >" + tmpDevice.getHealthTestStatus() + "<
" +
                    "Supplementary Fields Cnt >" + tmpKeyValueMap.Count + "<";
                for (int i = 0; i < tmpKeyValueMap.Count; i++)
                {
                    supplementaryFieldMsg = supplementaryFieldMsg +
                        "asObserverClass.onDeviceStateChange() - " +
                        "device >" + tmpDevice.getName() + "< supplementary key >"
+
                    tmpKeyValueMap.ElementAt(i).Key + "< value >" +
                    tmpKeyValueMap.ElementAt(i).Value + "<";
                }
                Console.WriteLine(msg);
                if (supplementaryFieldMsg.Length > 0)
                {
                    Console.WriteLine(supplementaryFieldMsg);
                }
            }

/*****************************************************************************
**********/
            public override void onDeviceDelete(netspire.Device device)
            {
                netspire.Device tmpDevice = new netspire.Device(device);
                netspire.KeyValueMap tmpKeyValueMap =
tmpDevice.getSupplementaryFields();
                String msg = "asObserverClass.onDeviceDelete() - Name >" +
tmpDevice.getName() + "< ";
                Console.WriteLine(msg);
            }
```

```
/*****************************************************************************
****************/
            public override void onVUMeterUpdate(int level)
            {
                String msg = "asObserverClass.onVUMeterUpdate() - " + level;
                Console.WriteLine(msg);
            }


/*****************************************************************************
****************/
            public override void onConfigUpdate(string progress, string key,
string value)
            {
             String msg = "asObserverClass.onConfigUpdate() - " + progress + ", "
             + key + ", " +
              value;
             Console.WriteLine(msg);
            }

/*****************************************************************************
***************/
            public override void onStateUpdate(bool isImportData, int key, string
dataMsg)
            {
             String msg = "asObserverClass.onStateUpdate() - " + isImportData +
             ", " + key + ", " + dataMsg;
                String msg1 = key + ", " + dataMsg;
                Console.WriteLine(msg1);
            }

/*****************************************************************************
*****************/
            public override void onUserDataUpdate(int elemId, int key, string
value)
            {
             String msg = "asObserverClass.onUserDataUpdate() - elemId: " +
             elemId + ", key: " + key + ", value: " + value;
                Console.WriteLine(msg);
            }

/*****************************************************************************
*******************/
            public override void onUserDataDelete(int elemId, int key)
            {
             String msg = "asObserverClass.onUserDataDelete() - elemId: " +
             elemId + ", key: " + key;
                Console.WriteLine(msg);
            }
        }

/*****************************************************************************
*****************
PAControllerObserver
******************************************************************************
****************/
        public class paObserverClass : netspire.PAControllerObserver
        {
```

```
/*******************************************************************************
****************/
            public paObserverClass()
            {
            }

/*******************************************************************************
****************/
            public override void onPaSourceUpdate(netspire.PaSource source)
            {
                netspire.PaSource tmpPaSource = new netspire.PaSource(source);
             String msg = "paObserverClass.onPaSourceUpdate() - sourceId >" +
             tmpPaSource.id + "< sourceState >" + tmpPaSource.state.ToString() +
             "<";
                Console.WriteLine(msg);
            }

/*******************************************************************************
*****************/
            public override void onPaSourceDelete(int sourceId)
            {
                String msg = "paObserverClass.onPaSourceDelete() - sourceId >" +
sourceId + "<";
                Console.WriteLine(msg);
            }

/*******************************************************************************
****************/
            public override void onPaSinkUpdate(netspire.PaSink sink)
            {
                netspire.PaSink tmpPaSink = new netspire.PaSink(sink);
                String msg = "paObserverClass.onPaSinkUpdate() - sinkId >" +
tmpPaSink.id + "< " +
                        "sinkState >" + tmpPaSink.state.ToString() + "< " +
                        "sinkAnnouncementType >" + tmpPaSink.announType.ToString()
+ "< " +
                        "sinkOutputGain >" + tmpPaSink.outputGain + "< " +
                        "sinkMuteActive >" + tmpPaSink.muteActive + "<";
                Console.WriteLine(msg);
            }


/*******************************************************************************
****************/
     public override void onPaSinkDelete(int sinkId)
            {
                String msg = "paObserverClass.onPaSinkDelete() - sinkId >" +
sinkId + "<";
                Console.WriteLine(msg);
            }

/*******************************************************************************
*****/
            public override void onPaTriggerUpdate(netspire.PaTrigger trigger)
            {
                netspire.PaTrigger tmpPaTrigger = new netspire.PaTrigger(trigger);
                String msg = "paObserverClass.onPaTriggerUpdate() - triggerId >" +
tmpPaTrigger.id + "< triggerState >" + tmpPaTrigger.state.ToString() + "<";
                Console.WriteLine(msg);
            }
/********************************************************************************
**********/
            public override void onPaTriggerDelete(int triggerId)
            {
                String msg = "paObserverClass.onPaTriggerDelete() - triggerId >" +
triggerId + "<";
                Console.WriteLine(msg);
            }


/********************************************************************************
************/
            public override void onPaSelectorUpdate(netspire.PaSelector selector)
            {
                netspire.PaSelector tmpPaSelector = new
netspire.PaSelector(selector);
                String msg = "paObserverClass.onPaSelectorUpdate() - selectorId >"
+ tmpPaSelector.id + "<";
                Console.WriteLine(msg);
            }


/********************************************************************************
*************/
            public override void onPaSelectorDelete(int selectorId)
            {
                String msg = "paObserverClass.onPaSelectorDelete() - selectorId >"
+ selectorId + "<";
                Console.WriteLine(msg);
            }
        }


/********************************************************************************
***************
*                                          CallControllerObserver
********************************************************************************
***************/
        public class callObserverClass : netspire.CallControllerObserver
        {

/********************************************************************************
***************/
            public callObserverClass()
            {
            }


/********************************************************************************
****************/
            public override void onCallUpdate(netspire.CallInfo callInfo)
            {
                netspire.CallInfo tmpCallInfo = new netspire.CallInfo(callInfo);
                String msg = "callObserverClass.onCallUpdate() - " +
             "callId >" + tmpCallInfo.callId + "< callStartTime >" +
             tmpCallInfo.callStartTime + "< " +
             "callDuration >" + tmpCallInfo.callDuration + "< callAnswerTime >" +
             tmpCallInfo.callAnswerTime + "< " +
             "callAPartyId >" + tmpCallInfo.callAPartyId + "< callBPartyId >" +
             tmpCallInfo.callBPartyId + "< callTargetIds >" +
             tmpCallInfo.callTargetIds + "< " +
             callState >" + tmpCallInfo.callState + "< callHangupCause >" +
             tmpCallInfo.callReleaseCause + "<";
                Console.WriteLine(msg);
            }

/*******************************************************************************
***************/
            public override void onCallDelete(int callId)
            {
                String msg = "callObserverClass.onCallDelete() - callId >" +
callId + "<";
                Console.WriteLine(msg);
            }


/*******************************************************************************
*************/
            public override void onCDRMessageUpdate(netspire.CDRInfo cdrInfo)
            {
                netspire.CDRInfo tmpCDRInfo = new netspire.CDRInfo(cdrInfo);
                String msg = "callObserverClass.onCDRMessageUpdate() - cdrId >" +
tmpCDRInfo.id + "<";
                Console.WriteLine(msg);
            }


/*******************************************************************************
***************/
            public override void onCDRMessageDelete(int cdrId)
            {
                String msg = "callObserverClass.onCDRMessageDelete() - cdrId >" +
cdrId + "<";
                Console.WriteLine(msg);
            }


/*******************************************************************************
****************/
            public override void onTerminalUpdate(netspire.TerminalInfo termInfo)
            {
                netspire.TerminalInfo tmpTerminalInfo = new
netspire.TerminalInfo(termInfo);
                String msg = "callObserverClass.onTerminalUpdate() - " +
                        "termId >" + tmpTerminalInfo.termId + "< termAddress >" +
tmpTerminalInfo.termAddress + "< " +
                        "termLocation > " + tmpTerminalInfo.termLocation + "<
termState >" + tmpTerminalInfo.termState + "<";
                Console.WriteLine(msg);
            }


/*******************************************************************************
***************/
            public override void onTerminalDelete(int termId)
            {
                String msg = "callObserverClass.onTerminalDelete() - termId >" +
termId + "<";
                Console.WriteLine(msg);
            }
        }
```

16.4 Sample Program – Update Service Information – C++

```
// NetSpire SDK Sample Program.
// Name:      Update Service Information.
// Description:
// Language   C++

// This sample application demonstrates how the automatic message
// generation functionality of the PassengerInformationController can be enabled
// using only limited prediction information, without creating a comprehensive
// and consistent Trip, TripStop and Service data structure.

// In the example below, only the predicted arrival and departure times for
// each station, the destination stop and the approaching/on platform state of
// services are provided to the PassengerInformationController.
// Providing additional information including delays, cancellation, and complete
// stopping pattern allows the system to display and announce messages that include
// the additional information.

// Standard Includes
#include <stdio.h>
#include <string>
#include <stdlib.h>
#include <iostream>
#include <ctime>

// Open Access Includes
#include "StandardIncludes.hpp"
#include "AudioServer.hpp"
#include "AudioServerObserver.hpp"


/****************************************************************************/
void CreateLines(PassengerInformationServer* piServer)
{
       // Create sample Lines - this is an optional step and line information
       // does not need to be set unless line specific behaviour is required.
       Line line1(5);
       line1.setName("City);
       piServer->addLine(line1);

       Line line2(12);
       line2.setName("Airport");
       piServer->addLine(line2);
}

/****************************************************************************/

void CreateVehicles(PassengerInformationServer* piServer)
{
       // Create vehicles - this is an optional step and vehicle information does
       // not need to be set unless real-time vehicle state is available.

       int vehicleIDs[] = { 101, 102, 103, 104, 105, 106, 107, 108, 109, 110 };

       for (int i = 0; i < sizeof(vehicleIDs) / sizeof (int); ++i)    // Assume 10
sample trains
       {
              int vehicleId = vehicleIDs[i];     // The vehicle ID would typically be
                                   // looked up from a list of available vehicles
```

```
			Vehicle vehicle(vehicleId);
			piServer->addVehicle(vehicle);
		}
}

/*****************************************************************************/

void CreateStationsAndPlatforms(PassengerInformationServer* piServer)
{
	// Create stations and platforms
	char* stationNames[] = { "Station1", "Station2", "Station3", "Station4" };
	int numStations = sizeof(stationNames) / sizeof(char*);

	for (int i = 0; i < numStations; ++i)
	{
		std::string name = stationNames[i];

		Station station(i, name, true);

		int platformNumbers[] = { 1, 2, 3, 4 };
		char* platformLocations[] = { "P1", "P2", "P3", "P4" };
		for (int j = 0; j < sizeof(platformNumbers) / sizeof(int); ++j)
		{
			int platformId = i * 10 + platformNumbers[j];
				// A unique number for each platform in the system

			PlatformInfo plat(platformId);

			std::string platLocation = name + platformLocations[j];
					// Typically looked up from a data store

			plat.setLocation(platLocation);
			station.setPlatform(platformId, plat);
		}

		piServer->addStation(station);
	}
}


/*****************************************************************************/
// Generates sample service information that cna be used without creating a
// comprehensive Trip, TripStop and Service data structure.
// The following function demonstrates how per platform prediction information
// can be sent to the PassengerInformationController to allow the system show
// and announce information on the next <numServices> services.
// In the example below, only the predicted arrival and departure times for
// each station, as well as the destination stop is provided to the
// PassengerInformationController. Providing additional information including
// delays, cancellation, and stopping pattern allows the system to display and
// announce messages that include this information.
std::list<Service> GenerateServices(int platformId, int numServices)
{
	static int lastService = 0;
	static int lastServiceStop = 0;
	int terminatingStops[] = { 4, 14, 24, 34 };
	time_t now = time(0);

	std::list<Service> nextServices;

	for (int i = 0; i < numServices; ++i)
	{
		Service service(lastService++);

		service.setLineId(12);      // optional
		service.setDirection(Trip::ASCENDING);    // optional
		service.setVehicleId(101 + i);     // optional
		service.setState(Service::RUNNING);

		if (i == 0)
		{
			// In this example, the first service is marked as
"APPROACHING_STATION".
			service.setState(Service::APPROACHING_STATION);
		}

		// Set the current and following ServiceStops - the last stop is used as
the destination station name
		std::list<ServiceStop> nextStops;

		// Current stop
		ServiceStop currentStop(lastServiceStop++);
		currentStop.setPlatformId(platformId);
		currentStop.setIsStopping(true);
		currentStop.setArvTime(now + (i * 240) + 60);
		currentStop.setDepTime(now + (i * 240) + 120);
		nextStops.push_back(currentStop);

		// Current stop is followe dby other stops in the stopping pattern.
		// This can be empty if stopping pattern display is not required.

		// Destination stop is the last entry of the stopping pattern. If the
		// stopping pattern contains the current
		// stop only, the current stop will be shown as the destination.
		ServiceStop destination(lastServiceStop++);
		destination.setPlatformId(terminatingStops[i]);
		destination.setIsStopping(true);
		// Arrival and departure time at destination are only required if
		// display of arrival/departure time at destination
		// is required.
		nextStops.push_back(destination);

		service.setStops(nextStops);
		nextServices.push_back(service);
	}

	return nextServices;
}


/*****************************************************************************/

void UpdateServiceLocations(PassengerInformationServer* piServer)
{
	std::vector<Station> stations = piServer->getStations();
	for (Station& station : stations)
	{
		std::vector<PlatformInfo> platforms = station.getListOfPlatforms();
		for (PlatformInfo& platform : platforms)
		{
			std::list<Service> nextServices =
			GenerateServices(platform.getId(), 3);
			platform.setPassers(nextServices);
```

```
                }
        }

}

/****************************************************************************/

void UpdatePIController(PassengerInformationServer* piServer)
{
        // Create Lines, Vehicles, Stations and Platforms.
        // These are typically static information and changes rarely.
        CreateLines(piServer);
        CreateVehicles(piServer);
        CreateStationsAndPlatforms(piServer);

        // Start automatic message updates
        piServer->enableAutomaticMessageGeneration();

        for (int count = 0; count < 100; ++ count)
        {
                // Send service arrival/departure prediction per platform
                UpdateServiceLocations(piServer);
#ifdef WIN32
                Sleep(60000);
#else
                sleep(60);
#endif
        }
}

/*****************************************************************************************/
void main (int argc, char *argv[])
{
    AudioServer theAudioServer;
    // setup connection parameters
    AudioServer::KeyValueMap connectionParametersKeyValuePair;
    AudioServer::StringArray audioserverAddresses;
    audioserverAddresses.push_back("10.1.40.100");
    audioserverAddresses.push_back("10.1.40.101");

    // connect to audio server
    theAudioServer.connect (audioserverAddresses, connectionParametersKeyValuePair);
    // wait until connection is established on the AudioServer. Alternatively do
something until 'onAudioConnected()' is not received.
    for (unsigned int i = 0; i < 1000; i++)
    {
        if (theAudioServer.isAudioConnected())
        {
            break;
        }
#ifdef WIN32
        Sleep(500);
#else
        sleep(1);
#endif
    }

    if (!theAudioServer.isAudioConnected())
    {
        std::cerr << "Cannot connect to the Audio Server." << std::endl;
        return EXIT_FAILURE;
    }
    //=============================================================================
    // Connection to audio server is established. Update Service, Vehicle,
    //  Station and
    // Platform information stored in the Passenger Information Server system.
    //=============================================================================
    UpdatePIController(theAudioServer.getPassengerInformationServer());

}
```