Eventide™
Communications

NexLog
Advanced Recording Solutions

DX
SERIES

Developer API Manual

Version 2025.1[6258]

P/N: #141367

P/N: #141367 Version 2025.1[6258]

Eventide Communications LLC
One Alsan Way
Little Ferry, NJ 07643
201-641-1200

www.eventidecommunications.com

Table Of Contents

2. Associating Metadata via NexLog Metadata Feeds

3. Open Database Connectivity (ODBC)

4. Accessing Eventide NexLog via SOAP

5. Interfacing to NexLog's REST API

6. NexLog Generic CAD API

7. Reporting Problems

NEXLOG APPLICATION PROGRAMING INTERFACES

NexLog Recorders support several methods of accessing media and metadata via a variety of programmable APIs, including HTTP, Metadata Feeds, ODBC, SNMP, CAD, and SOAP. Use of Eventide APIs may be subject to purchase of add-on licenses per NexLog recorder, please contact service@eventidecommunications.com to discuss your needs.

1. ACCESSING MEDIA VIA HTTP(S)

NexLog recorders support several methods of accessing media via a programmable API. This section describes accessing media via an HTTP interface. Media can be retrieved either as a single record or as a time range from a list of physical channels. In order to retrieve a single media file, the user must have information about the call that's being pulled. This can either be the CallGuid or some other metadata associated with another system. For example, if a call was annotated with a unique value from an external source using the Eventide NexLog Metadata Feeds protocol then the media could be retrieved using that known value.

The HTTP protocol requires Basic web authentication. Credentials can be sent in the URL or in response to an Authentication request. The recorder can also be configured to support or require SSL (See NexLog User Manual). The authenticating user must have the appropriate channel/resource permissions to access the requested recordings. By default, an admin user will have permissions to all channels/resources. See the NexLog User Manual for information on managing user permissions.

Unless otherwise specified, Media is returned as WAV files containing linear PCM 16-bit signed samples with a sampling rate of 8000 hertz. If requesting RTP streaming the recommendation is to request 8-bit Mulaw compression by adding comp=1 to the GET request.

Media is requested via the URL, <PROTOCOL><ADDRESS>/agent/retrieveMedia.py

The request must contain a valid URL argument set. Note that the URI arguments must be properly encoded. Examples of valid URLs can be found here

GET /agent/retrieveMedia.py

1.1. URL Parameters - General

- **c** - Specify which channel(s) will be used in the command in the form c=<channel1>[,<channel2>,...]
- **comp** - Audio compression. For RTP streaming this should always be comp=1 for 8-bit G.711 Mulaw. For WAV file download do not specify the compression type to download the default uncompressed 16-bit linear audio
- **ctw** - Send audio back as a WAV file. Including ctw=1 will send the HTTP Content-Type header "audio/x-wav" for the client to play the audio as a WAV. The header "application/octet-stream" will present the audio as a downloadable file

- **d** - Enable debug. d=1 enables output to the debug log which is accessible at <PROTOCOL><ADDRESS>/downloads/debug.txt
- **k** - Use a stored metadata key to find a specific recording. This must be used with the value command v to specify a unique key-value pair to identify a single recording
- **l** - Local playback. Play through the front panel speaker of the recorder
- **m** - Mono audio output. m=1 should always be set for RTP streaming
- **q** - Run the database query, but do not send audio
- **skipSilence** - Skip the silent spaces between calls. Example, skipSilence=1
- **t1** - The start time in UNIX format for either the dataset window or the playback start time
- **t1iso** - The start time in ISO format for either the dataset window or the playback start time
- **t2** - The end time in UNIX format for either the dataset window or the playback end time
- **t2iso** - The end time in ISO format for either the dataset window or the playback end time
- **v** - Use a stored metadata value to find a specific recording. This must be used with the key command k to specify a unique key-value pair to identify a single recording
- **w** - Wait for completion before sending to client, `0` (default), `1`

1.2. URL parameters - RTP specific

- **buffer** - Number of seconds to stream ahead of the current time. For example, buffer=1 will send one second of RTP data as quickly as possible and then subsequent RTP packets will be sent at the same speed that they were recorded.
- **cached** - Start playback from a cached dataset. Playback should start in near real-time. This should be used after the dataset is created using the prepare and session commands.
- **clearSession** - Stop playback and free the prepared dataset.
- **multicast** - Send RTP to multicast destination. Used with the rtp command to specify the IP address and port.
- **noWait** - Run the request in the background and return immediately. Making requests with noWait=1 does not provide feedback as to the success or failure of the requested action. This can be used when starting streaming so that a TCP connection does not have to remain open during the duration of the audio playback.
- **payload_ms** - Specify the frame size of the RTP audio packets. This command should be used for any receiving party that requires a specific frame size for incoming RTP audio. Sending the command payload_ms=20 with the streaming request will result in the recorder sending 20 milliseconds of audio in each RTP packet. The default frame size is 60ms.
- **prepare** - Prepare a dataset for playback. The preparation may take many seconds to complete when long periods of time or many channels are requested. Once the dataset is prepared, subsequent playback requests using the cached dataset should start immediately. A dataset stays

prepared until a clearSession=1 is called for that session or 30 days elapse since the creation of the dataset. The session may only have one dataset prepared at a time. A channel, playback start time, and stop time must be specified in the playback command URL. The channel argument will filter playback from within the prepared dataset. The start and stop time will be used to start and stop playback within the prepared dataset.

- **rtp** - Comma separated list of destination address and port to stream RTP. Defined as rtp=<address>:<port>[,...]
- **session** - Used to identify a requesting client such that a dataset (see 'prepare') and playback can be referenced in corresponding calls. The session is defined by the calling process. Sessions can be re-used. A session should not have more than one stream active at a time. Note, if more than one stream is active for a session it won't be possible to stop playback. Example, session=cwp1
- **speed** - Controls the speed of playback. This is a value from -100 to 100 with 0 being normal speed. Negative numbers are slower and positive numbers are faster. Example, speed=-50
- **stopPlayback** - Stops the current playback for the specified session. Example, stopPlayback=1

1.3. URL Examples

The HTTP GET request must contain a valid URL argument set. Note that the URI arguments must be properly encoded. Examples of valid URLs are as follows.

1.3.1. Retrieve media with a matching value for a metadata field

Example:

```
GET /agent/retrieveMedia.py?k=MyCallId&v=111222333 HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: */*
Authorization: Basic RXZlbnRpZGU6MTIzNDU=
```

Description:

Stream the WAV file for a call record containing the value of 111222333 for the metadata field named MyCallId. The metadata field may be a field created by the recorder during normal recording operations (SIP CALLID), or a custom field entered by a 3rd party and populated from an external source. Note that the specified key column name should be indexed for best performance.

1.3.2. Retrieve media by matching an Eventide CallGuid

Example:

```
GET /agent/retrieveMedia.py?v=ATG83KsaIoJ91NaW HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: */*
Authorization: Basic RXZlbnRpZGU6MTIzNDU=
```

Description:

Stream the WAV file for the call with CallGuid value ATG83KsaIoJ91NaW. The CallGuid must be determined using a different API (ODBC, SOAP, REST).

1.3.3. Retrieve media by matching an Eventide ID

Example:

```
GET /agent/retrieveMedia.py?k=id&v=1 HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: */*
Authorization: Basic RXZlbnRpZGU6MTIzNDU=
```

Description:

Stream the WAV file that matches ID equal to 1. The recorder has a sequenced ID field. The first call on the recorder will have id=1. Note that when the recorder runs out of space it will remove the oldest call ID's to make room for new ID's.

1.3.4. Retrieve media via complex date and channel based criteria

This example shows media retrieval based on a start time and an end time and a list of channels. Times are specified as either an ISO date time (`t1iso` , `t2iso`) string or a UNIX date time (`t1` , `t2`).

Example:

```
GET /agent/retrieveMedia.py?
c=1,2&t1iso=20121023T040200&t2iso=20121023T050200 HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: */*
Authorization: Basic RXZlbnRpZGU6MTIzNDU=
```

Description:

Create a WAV file with recordings between two dates on physical channels 1 and 2 where the recordings play in real-time. Silence will be included in the file. Note that only authorized recordings will be included in the file.

1.3.5. Retrieve media packaged as RTP

By default, the media is returned to the requester in the resultant payload of the HTTP GET command. The API can also request that the resulting media is streamed from the recorder to specified addresses. The stream consists of Real-Time Transport Protocol (RTP) packetized media sent over the UDP protocol.

When RTP mode is specified in the HTTP GET request via the rtp parameter, the RTP audio will be streamed to the specified addresses, and status text will be returned from the GET request. Each status text is separated by a newline character. The current playback time is sent out at least one time per second for a play request that is not backgrounded (without noWait=1). Status messages that begin with “Success.” Indicate normal operation. Messages that begin with “Warning.” Indicate that the controlling program might need to handle a problem. Messages that begin with “Error.” Indicate that there is a problem that might require the attention of the end user. Possible status messages include, but are not limited to,

Success. Command sent. - This will be seen when noWait=1 is sent. This should only be used on playback of streaming RTP when the HTTP GET command cannot be held open until the end of the audio playback.

Success. Streaming started. - This text is sent as RTP streaming starts and is followed by update messages (see “Sent=).

Sent=(seconds from beginning of file) - This update is sent in the form “Sent= 54.375” every 0.125 seconds during the duration of RTP streaming unless the play command was started with noWait=1.

Streaming complete. Sent bytes=(number of bytes sent) - This is the final message during RTP streaming when the playback reaches the end of the time window specified.

Warning. Failed to find stream. - This indicates that the recorder is not able to find the named session. Either the named session has not been created, has already been removed with clearSession=1, or the session has expired 30 days after creation.

Success. Streaming stopped. - This message is returned when RTP streaming is in progress and the stopPlayback=1 command is sent.

Success. Cache removed. - This message is returned when the clearSession=1 command is sent and the named session currently exists.

Warning. Cache not found. - This message is returned when the clearSession=1 command is sent and the named session does not exist or has already been cleared.

Success. Dataset ready. - This message is returned when running a prepare=1 command to set up the dataset window with start time, stop time, and channels queried from the database and saved in a cache file for faster access.

Error. Database connection failed for user, (username). - The user credentials passed with the HTTP GET request in the authentication header must have rights to query the database or this error will be generated.

Error. Messaging failed for user, (username). - The messaging system internal to the recorder returned an error for the authenticated user.

Error. Init failed. - Internal error indicating that the streaming process failed to initialize. Possible reasons include a failure to connect to the internal media server or database.

Error. Failed to find record for key=(key) value=(value)". The record may not exist or you may not have permission to access it. - No records were returned when the database was queried for the given key and value pair.

Error. Permission to access this recording has been denied. - The user permissions on the recorder did not allow access to the specified recording with the credentials that were passed in with the authentication header of the HTTP GET command.

Error. Media type not supported. - This is likely due to attempting to stream a screen capture recording.

Error. Failed to find database cache file. - This error can occur if an attempt is made to play from a session that has been destroyed or never created.

Error. Media not processed. - No audio was generated for the request. This could indicate invalid time parameters being sent.

1.3.5.1. Streaming notes

The HTTP GET command must be kept open for the duration of the request unless processing is backgrounded with noWait=1. If it is closed prematurely then streaming will stop.

The amount of time between the initial request and the start of RTP streaming is not guaranteed. To achieve synchronization between various media sources it is necessary to process the status command from the HTTP GET request. Using the prepare=1 and cached=1 commands will make the playback timing much more predictable.

Format <PROTOCOL><ADDRESS>/agent/retrieveMedia.py?c=<channel numbers>&t1iso=<ISO formatted time>&t2iso=<ISO formatted time>&rtp=<DESTINATION_ADDRESS>:<DESTINATION_PORT>&buffer=<seconds to buffer>

Example: http://192.168.1.1/agent/retrieveMedia.py?c=1,2&t1iso=20121023T040200&t2iso=20121023T050200&rtp=192.168.1.1:6777&buffer=0

Description: Start streaming recordings between two dates on physical channels 1 and 2. Silence will be included in the file to create an accurate representation of events. The recordings will be streamed as RTP to 192.168.1.1:6777. The data will be sent out in real-time without any buffered audio.

1.3.5.2. Example session to create a dataset and play from it skipping silence

Initialize the dataset named sess1 and include channels 1-6 from midnight 25 October 2019 to 23:59:59 on 25 October 2019. http://192.168.1.1/agent/retrieveMedia.py?c=1,2,3,4,5,6&comp=1&m=1&prepare=1&session=sess1&t1iso=20191025T000000&t2iso=20191025T235959
Success. Dataset ready.

Play channels 1-6 mixed together from times 20:08:00 to 20:08:03.

http://192.168.1.1/agent/retrieveMedia.py?buffer=1&cached=1&c=1,2,3,4,5,6&comp=1&m=1&skipSilence=1&rtp=192.168.1.110:6777&session=sess1&

Success. Streaming started.

Sent= 0.25

Sent= 0.375

Sent= 0.5

Sent= 0.625

Sent= 0.75

Sent= 0.875

Sent= 1.0

Sent= 1.125

Sent= 1.25

Sent= 1.375

Sent= 1.5

Sent= 1.625

Sent= 1.75

Sent= 1.875

Sent= 2.0

Streaming complete. Sent bytes=16442

Attempting to stop playback of a session that has already stopped.

http://192.168.1.1/agent/retrieveMedia.py?session=sess1&stopPlayback=1

Warning. Failed to find stream.

Play only channel 1 from times 20:08:00 to 20:08:01.

http://192.168.1.1/agent/retrieveMedia.py?
buffer=1&cached=1&c=1&comp=1&m=1&rtp=192.168.1.110:6777&session=sess1&t1iso=20191025T200800&t2

Success. Streaming started.

Sent= 0.25

Sent= 0.375

Sent= 0.5

Sent= 0.625

Sent= 0.75

Sent= 0.875

Sent= 1.0

Sent= 1.125

Sent= 1.25

Sent= 1.375

Sent= 1.5

Sent= 1.625

Sent= 1.75

Sent= 1.875

Sent= 2.0

Sent= 2.05525

Streaming complete. Sent bytes=16442

Delete the session.

http://192.168.1.1/agent/retrieveMedia.py?clearSession=1&session=sess1

Success. Cache removed.

This page was intentionally left 99.88% blank.

2. ASSOCIATING METADATA VIA NEXLOG METADATA FEEDS

2.1. Metadata Feeds

NexLog recorders support Metadata Feeds, which are used to send control data to the recorder. The format of these feeds and the behavior the recorder will take upon receiving commands via these feeds is programmable in the Recorder's Configuration Manager under System ->Configuration Files. The Metadata Feed parser is highly configurable and often used to parse ANI/ALI or SMDR data in a fixed format. However it can also be configured to parse a simple set of commands received as ASCII text over a UDP or TCP port. This document will focus on this use of the Metadata Feed Parser.

Metadata feeds are a licensed feature on the recorder and a license key must be programmed into the recorder to activate the feed. The charge for the license key may also include assistance from an Eventide Integrations engineer to program the recorder's metadata feed format and behavior to match the customer's needs. Please contact Eventide Sales for information on pricing. A metadata feed can be configured to take commands from the recorder's built-in serial ports or an add-on serial port board available from Eventide, or from a UDP or TCP connection over the recorder's Ethernet port. Commands are sent to the recorder using a simple unidirectional ASCII-based protocol, and the format is configurable. Upon receiving a message via this connection, the recorder can be configured to perform a command on a given channel; for example, "Start a call if not already recording," "Stop a call that is in progress," "Stop the current call and start a new one," or "Delete the call in progress." In addition, the metadata feed can send additional information about the call in-progress to the recorder to be stored in the recorder database along with the other normally stored information. This information can be viewed or queried using Eventide's graphical retrieval clients (MediaWorks, MediaAgent). It can also be accessed programmatically over ODBC. For example, this can be used to have an external controlling program start a call and give it a known identifier so that the call can later be retrieved by that identifier.

The examples below describe one simple command set which can be programmed into the Metadata Feed Parser. Different command formats and semantics can also be programmed for cases where the remote software sending commands to the recorder cannot be modified. For new development or integrations, these standard commands normally suffice for most needs. From this point forward the information in this section refers only to this single possible command set configuration.

Commands will be sent to the recorder from remote software by sending a UDP/TCP command packet to the recorder on port 5000. The data is sent as an ASCII text string starting with the string '<<' and ending with the string '>>'. Individual fields are separated with the string '::'. A newline (\n) may be sent between commands and will be ignored. Commands are executed by the recorder as they arrive.

2.2. Metadata Commands

2.2.1. Start

Start recording on a given physical recorder channel.

Format:

```
<<CHANNEL_NUMBER::START>>
```

Example:

```
<<61::START>>
```

Description:

When this command is received, the recorder will start recording on channel 61. If a call is already in progress on 61, nothing will be changed, and the current call in progress will continue to record.

2.2.2. Stop

Stop recording on a given physical recorder channel.

Format:

```
<<CHANNEL_NUMBER::STOP>>
```

Example:

```
<<54::STOP>>
```

Description:

When this command is received, the recorder will stop recording on channel 54. If no call is in progress on 54, nothing will be changed.

2.2.3. Break

Perform a call break on a physical recorder channel.

Format:

```
<<CHANNEL_NUMBER::BREAK>>
```

Example:

```
<<36::BREAK>>
```

Description:

When this command is received, if no call is in progress on channel 36, the recorder will begin recording a new call. If a call is already in progress on channel 36, that call will be terminated and a new call started.

2.2.4. Delete

Delete the current call in progress or previous call on a physical recorder channel.

Format:

```
<<CHANNEL_NUMBER::DELETE>>
```

Example:

```
<<24::DELETE>>
```

Description:

When this command is received, if no call is in progress on channel 24, the previous call on the channel will be deleted. If a call is in progress, it will continue to record until it is stopped, at which point it will be purged from the recorder's call database within the next few minutes. Note that unless this API command is specifically requested by the customer, it will not be programmed in or available on the recorder.

2.2.5. Start (with Metadata)

Start recording on a given recorder channel and apply metadata to it.

Format:

```
<<CHANNEL_NUMBER::START::METADATA KEY::METADATA VALUE>>
```

Example:

```
<<16::START::CUSTOMER_NAME::Eventide, Inc.>>
```

Description:

When this command is received, if no call is in progress on channel 16, a new call is started, if a call is already in progress, it is allowed to continue. In addition, regardless of whether a new call is started or not, the value sent in "Metadata Value" will be written to the database field "Metadata Key" in the database record for the call. The Metadata field specified by "Metadata Key" must have already been created in the Recorder's Configuration Manager under Recording -> Custom Fields, and be of a data type compatible with the value sent as "Metadata Value". The Text sent as the Metadata Key must match exactly the name of one of the configured Custom Fields.

2.2.6. Stop (with Metadata)

Apply metadata to a call on a given channel and stop it.

Format:

```
<<CHANNEL_NUMBER::STOP::METADATA KEY::METADATA VALUE>>
```

Example:

```
<<8::STOP::CUSTOMER_NAME::Eventide, Inc.>>
```

Description:

If no call is in progress on channel 8, the command is ignored; otherwise, the metadata is applied and the call stopped. See Start (with Metadata) for more info about the Metadata Key and Value fields.

2.2.7. Break_Then_Apply (Metadata)

Break the current call, start a new one, and apply metadata to it.

Format:

```
<<CHANNEL_NUMBER::BREAK_THEN_APPLY::METADATA KEY::METADATA VALUE>>
```

Example:

```
<<125::BREAK_THEN_APPLY::CUSTOMER_NAME::Eventide, Inc.>>
```

Description:

If a call is in progress on the channel it will be stopped. Regardless of whether a call was in progress or not, a new call will then be started and the metadata included in the command string will be applied to it.

2.2.8. None

Apply metadata to the current/last call and perform no additional actions.

Format:

```
<<CHANNEL_NUMBER::NONE::METADATA KEY::METADATA VALUE>>
```

Example:

```
<<34::NONE::Foo::Bar>>
```

Description:

If no call is in progress, the metadata is applied to the most recent call that took place on channel 34. If no call has taken place on channel 34 since recorder start-up, the metadata is ignored. If a call is currently in progress, the metadata is applied to that call. No call stops or starts take place. This command can be sent multiple times with different Key/value pairs to set multiple custom fields.

2.2.9. Cache

Apply Metadata to the current/next call and perform no additional action.

Format:

```
<<CHANNEL_NUMBER::CACHE::METADATA KEY::METADATA VALUE>>
```

Example:

```
<<45::CACHE::Foo::Bar>>
```

Description:

If a call is currently in progress on the channel, the metadata in the command string is applied to it. If no call is currently in progress, the metadata is cached. The next call to start on this channel will get the metadata applied to it. If no more calls start on this channel before the recorder is powered off, the metadata will be discarded. In addition, if another CACHE command string is received by the recorder for this channel before call start, the old cached metadata will be discarded.

2.3. Metadata Notes

1. In addition to calls being started or stopped by the external program, the recorder channels will still respond to start/stop events from other configured methods. For example, it is acceptable to configure a channel to have calls starting and stopping due to VOX call detection and also provide additional start/stop events. Normally however, the recorder channels to be externally controlled will be configured to ignore all other call start sources and to obey only external sources, or they will be configured to use their normal internal methods for start/stop control, and the metadata feed will only be used to apply metadata to the calls. In some applications though, the hybrid approach is desirable and is available. In other systems, only CACHE or NONE events are sent to apply metadata to calls started and stopped via the standard VOX or signaling interfaces.
2. All command strings above are available with analog call sources. There is one caveat for digital call sources (T1/E1, ISDNBRI, PBX, etc.). Unlike an analog source, which always has data available to record (though it may simply be silence), on digital sources no data is available to record if no call is being sent along the signal path. Therefore, forcing a call start on a digital channel may not perform as expected if no corresponding signal is available to record. To work around this issue, the external application should only send a call start for a channel if it knows audio data is currently being sent on that channel. In addition, for these digital channels, it is preferable to send START events without metadata and then send the metadata via a NONE or CACHED event a few seconds later. This is because the recorder cannot force an immediate call start to make sure there is a call to which the metadata should be applied. This method allows the recorder to request a call start and wait for the data to begin to flow, and then apply the metadata when data is available. With analog sources, this is not necessary because the data is always “flowing” into the system regardless of the call states. With Digital sources, typically only the NONE, CACHE and DELETE commands are used and call control is based on presence or absence of data.
3. The command formats above are the defaults supplied by Eventide, however the formats can be altered and some of the behavior of the commands themselves can be altered.

This page was intentionally left 99.88% blank.

3. OPEN DATABASE CONNECTIVITY (ODBC)

3.1. Overview

This document describes how to install the PostgreSQL ODBC driver, connect to the PostgreSQL database residing in a NexLog recording system, and generate a customized report of the data contained therein.

You'll learn how to create a user with permissions to the Eventide database schema, select individual tables for inclusion in the report, and how to design a report using Microsoft Office Excel and Access.

Note: This exercise requires that you have Microsoft Office 2010 or higher and you are running Microsoft Windows 7 (32-bit or 64-bit). Your recorder must be configured correctly in the network. This exercise requires administrator access to create or change user rights. It also involves working with incidents and data from an Eventide NexLog recorder; make sure that you have at least one incident for testing. Use Eventide MediaWorks DX when working with incidents.

At the end of this document there is a list of all accessible PostgreSQL tables and views with descriptions of their fields.

3.2. Installing the ODBC Driver

ODBC is an acronym for Open Database Connectivity, a standard, widely adopted method for accessing databases. The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS can accurately interpret.

The Eventide NexLog series of call logging recorders employs PostgreSQL, an open system, Linux-based DBMS. In order for you to access the database from Microsoft Windows, you must first install the PostgreSQL ODBC driver.

To do this, simply download, and install the version that works with your operating system from PostgreSQL. The files can be located from the source at:

http://www.postgresql.org/ftp/odbc/versions/msi/

Make sure that you install the 32- or 64-bit version based on which version of Windows 7 you are using. After the installation is complete you will need to create the connection to the recorder. First though, you must create a user that has the necessary rights to connect.

3.3. Adding a User with Database Access Permissions

You must have administrator rights to create users and change permissions in the recorder. The following steps can be done using the front panel of the recorder or using the Configuration Manager portal of the recorder.

- Log on to the recorder with administrator rights.
- Under Setup locate Users and Security.
- Click on Users.

> **Note**
>
> By default, the Eventide user has full administrative rights, and although you may use this account to establish the connection it is recommended to use a restricted account.

- Click on Add User provide a name, and a password. Note: usernames and passwords are case sensitive.
- The user must be part of the Researchers permission group.
- User information can be left as defaults, and the account must be enabled. If there are password expiration policies, set this password to never expire or make a note of the frequency for changing the passwords in the ODBC configuration page.
- At this time you can exit out of the Setup menu.

USERNAME	ADMIN	GROUPS	ACCOUNT STATUS
Eventide	Yes	All	Enabled
odbc	No	Researchers	Enabled

Fig. 3.1 **User Security Page on the Front Panel**

3.4. Establishing an ODBC Database Connection

After the installation of the PostgreSQL ODBC driver, you must create a connection within Windows.

In this document we will use Windows 7 64-bit, but the configurations are identical for the 32-bit version.

- Open Control Panel, locate the Administrative Tools
- Open Data Sources (OBDC); you may also use the RUN command: odbcad32.exe
- In the ODBC Data Source Administrator windows select Add.
- There are two versions ANSI, or Unicode. For this test we will use Unicode.
- Select PostgreSQL Unicode(x64) from the list, then Finish.

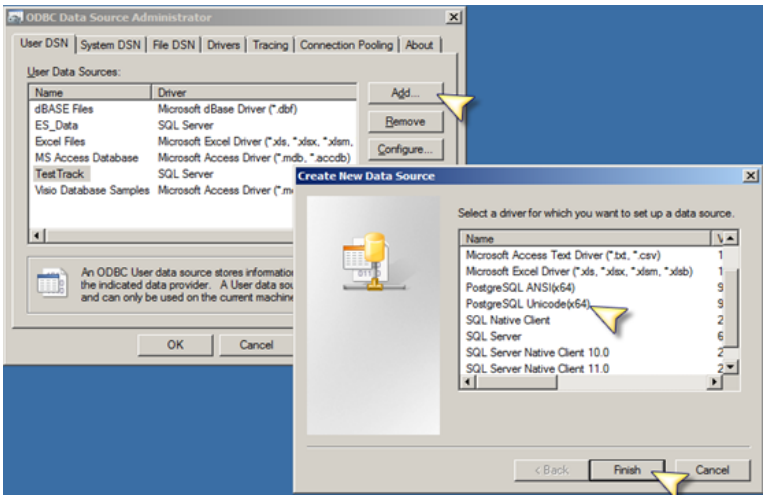

Fig. 3.2 **Add PostgreSQL Connector**

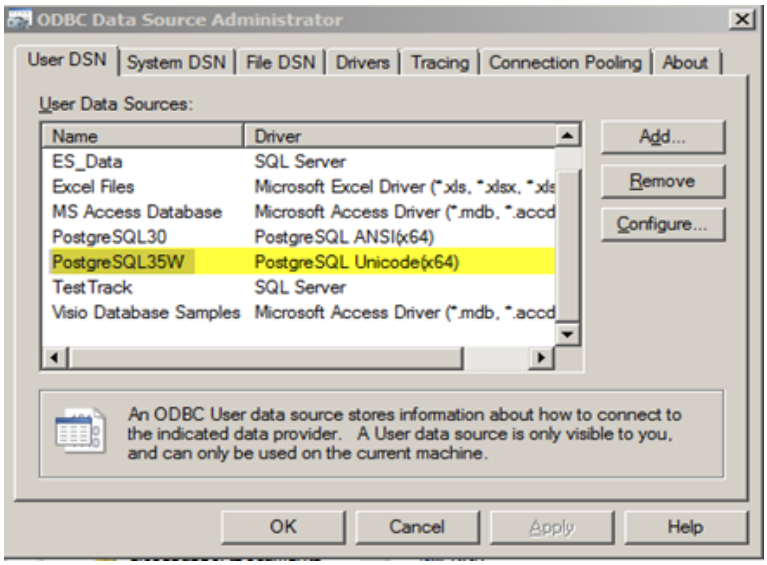

Fig. 3.3 **ODBC Window**

Select PostgreSQL Unicode, then configure using the following settings: (see Fig. 3.4)

- **Datasource:** meaningful name for this connection
- **Database:** lj (case sensitive)
- **SSL Mode:** Disable
- **Server:** IP address of the recorder
- **Port:** 5432
- **User Name:** From Adding Database Access Permissions
- **Password:** From Adding Database Access Permissions

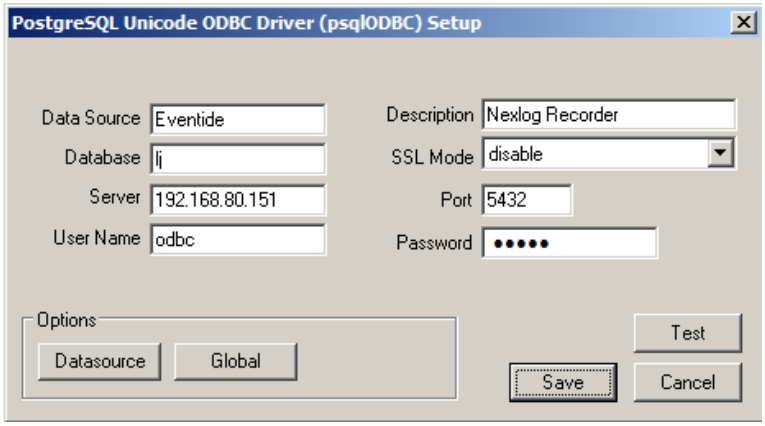

Fig. 3.4 **ODBC Configuration**

After completing the steps above test the connection. The connection test should show successful. Then press OK, save the connection, and close all other windows.

3.5. Setting up a Connection using Microsoft Excel

The following procedures may help to connect other software products that support OBDC data querying.

Launch Microsoft Excel, this document uses version 2010. Open a new book, from the ribbon menu select Data, and from the ribbon select from Other Sources.

Other Sources will provide a drop down you have two options. Data Connection Wizard, or Microsoft Query. This example will take a look at the Microsoft Query Wizard.

You may add tables to the report by means of selecting them from the list of available tables. In the example below we took the following tables:

'v_incident', 'callincidentgrouping', 'v_call', 'v_record'

The unique identifier for the tables 'callincidentgrouping', 'v_call', 'v_record' is the 'callguid'.

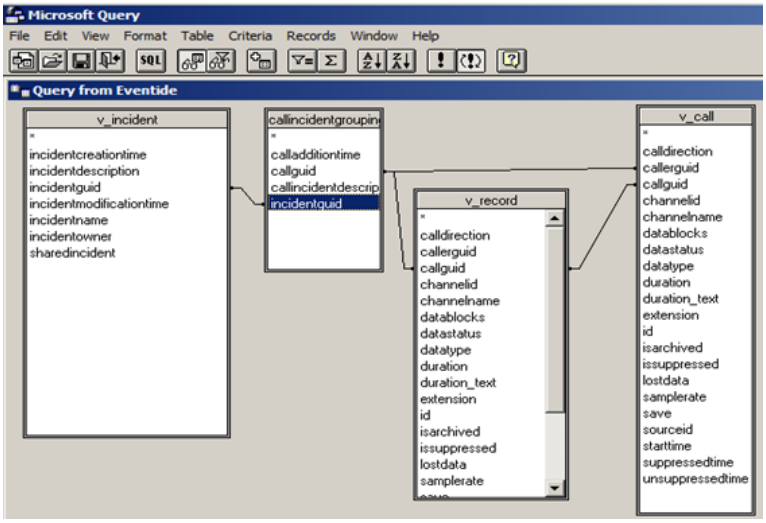

Fig. 3.5 **Excel Query Editor**

3.5.1. V_RECORD

This view provides detail information of all call records on the system.

Column Name	Field Type	Width	Description
callguid	character	16	
channelid	integer		
datatype	smallint		Compression Type
calldirection	character	1	I for Incoming, O for Outgoing, U for Unknown.
callerguid	character	16	Custom.
datastatus	character	1	C for Complete, P for In Progress, R for Removed, I for Partial Removed.
isarchived	boolean		Custom.
issuppressed	boolean		Was the call audio for this call Suppressed?
save	boolean		Is this call marked Protected.
extension	integer		Custom.
starttime_timestamp	timestamp without time zone		
stoptime_timestamp	timestamp without time zone		
duration	integer		Duration in seconds.
duration_text	interval		Interval datatype for duration.
lostdata	integer		Custom.

Column Name	Field Type	Width	Description
datablocks	integer		Mediasize in kilobytes; one block is 1k of data.
samplerate	integer		Sample rate of recording.
suppresedtime	integer		
unsuppressedtime	integer		
sourceid	integer		
channelname	text		
id	integer		Database record ID.
callguid	character	16	Custom; will be blank if there is no metadata associated with the call.
dtmf	text		DTMF
calling_party	text		
caller_id	text		Caller ID.
annotations	text		Stored in XML format.

V_Record will also include any Custom Fields, such as Location, Speech, Agent_ID, User_ID, etc., that are configured on the recorder.

3.5.2. V_ALERTHISTORY

This view provides detail information of all system alerts.

Column Name	Field Type	Width	Description
alertcode	Integer		Numerical value that represents the alert code.
eventtime	Timestamp		The time the alert occurred

Column Name	Field Type	Width	Description
serial	Serial		Auto-incrementing field
alertguid	Character	16	A guide used by the system to track alerts
displaytext	Character	512	The text message that displays in the alert
isacknowledged	Integer		Has this alert been acknowledge
timeacknowledged	Integer		The time the alert was acknowledge
acknowledginguser	Character	63	The user that acknowledge the alert
acknowledgingprocess	Character	63	The process used to acknowledge the alert
isresolved	Integer		Has this alert been resolved. Only used if the alert requires resolution
resolveddisplaytext	Character	512	Message that is displayed once the alert is resolved
timeresolved	Timestamp		Time the alert was resolved
resolvinguser	Character	63	User that resolved the alert
resolvingprocess	Character	63	The process used to resolve the alert
triggeringprocessname	Character	63	The process that triggered the alert
triggeringusername	Character	63	The user that triggered the alert

3.5.3. V_DAILYSTATISTICS

This view provides detail information of daily system statistics.

Column Name	Field Type	Width	Description
Id	Integer		Auto-incrementing field
datetime	Timestamp		Date and time without time zone

Column Name	Field Type	Width	Description
callcount	Integer		Record count from date time
callcountsincelast	Integer		Difference between the call count and the
displaytext	Character	512	The text message that displays in the alert

This page was intentionally left 99.88% blank.

4. ACCESSING EVENTIDE NEXLOG VIA SOAP

4.1. Overview

This document requires you have the SOAP API Example Application (Part Number: 141250-01) that demonstrates access to Eventide's NexLog server via Simple Object Access Protocol (SOAP) using a .NET client. If you do not, contact service@eventidecommunications.com to request this application.

The application demonstrates the following features:

- Logging into NexLog
- Retrieving channel names
- Retrieving channel information
- Retrieving call history
- Retrieving call metadata
- Starting and stopping recording
- Squashing a recording
- Setting channel metadata

4.2. Setting up Visual Studio 2012

The Demo was built as a WPF (Windows Presentation Foundation) application using Visual Studio Professional 2012. The only additional download required was the Extended WPF Tookit which provides the DateTimePicker control. This is available as described in the Supporting Information section, or may be downloaded via the NuGet Package Manager.

The WSDL is loaded into the project as a Service Reference. To create a Service Reference for your own project you can use the WSDL link in the Supporting Information section. It can be loaded into VS2012 via **Project|Add Service Reference**, and enter the link into the Address field.

The Demo is written in C#.

4.3. Source Files

MainWindow.xaml.cs

This module is created by VS when the project is created. It sets up the UI and is the coordinating code for the application.

Login.cs

This module manages the login dialog that appears when you start the app. It creates the client object which represents the WSDL document as a service reference.

SetMetadata.xaml.cs

This module manages the 'Set Channel Metadata' window.

NexLog.cs

This module encapsulates the service reference and is the only direct user of the reference. It provides methods for use by the other classes.

CookieBehavior.cs

This module encapsulates the manual re-injection of the session cookie into each outgoing HTTP packet. It is reusable and may be considered opaque after its initialization.

4.4. Expectations

The Expect100Continue setting configures how the HTTP session is handled. For details on this see: http://msdn.microsoft.com/en-us/library/system.net.servicepointmanager.expect100continue.aspx

The setting must be false to achieve compatibility with NexLog.

```
// Code Snippet
System.Net.ServicePointManager.Expect100Continue = false;
// disable expectations
```

4.5. Cookie Handling

The NexLog requires a session cookie to maintain the integrity of the session. By default, WPF applications do not handle cookie management automatically; the cookie has to be inserted into the outgoing HTTP. This functionality is encapsulated in class CookieBehavior. If this module is not installed then all SOAP operations after the initial login will fail.

```
// Code Snippet:
// See the CookieBehavior.cs module for details
_client.Endpoint.EndpointBehaviors.Add(new CookieBehavior());
```

4.6. Logging In

The code snippet shows the login method and the returned objects sessionKey and passwordExpire. Both of these are both unused as the CookieBehavior class handles the session key.

```
// Code Snippet
_client.login(txtUsername.Text, // text from controls
   txtPassword.Text,
   txtIPAddress.Text,
   out sessionKey, // unused
   out passwordExpire); // unused
```

4.7. Retrieving Channel Names

The getAllChannel() call returns an array of ChannelEntity objects, each describing a recording channel.

```
// Code Snippet
// Retrieve the channel names from NexLog
//
ServiceReference2.ChannelEntity[] channels = _client.getAllChannel();
foreach (ServiceReference2.ChannelEntity channel in channels)
{
_channelEntity[channel.name] = channel;
}
```

See the ChannelEntity object definition for a full description of the channel. A channel may be analog (ie, accepting an interface with POTS), or digital. This characteristic defines what the channel can and cannot do, and this will be outlined later in this document.

Note

NexLog systems with a large number of channels (>32) may not be queried in this way because the returned data can exceed the maximum 64K message size specified by WPF. For details on this, check the following link: http://blogs.msdn.com/b/drnick/archive/2006/03/10/547568.aspx

You can adjust the MaxBufferSize parameter or use the following code. Be aware, though, that the getChannelCount and getChannelByIndex calls are not implemented in all versions of NexLog.

```
// Code snippet
for (int ii = 1; ii <= _client.getChannelCount(); ii++)
{
ServiceReference2.ChannelEntity channel =
_client.getChannelByIndex(ii);
_channelEntity[channel.name] = channel;
}
```

4.8. Retrieving Call Data

A call is encapsulated by the MediaRecordEntity object. When a new channel is selected in the application it submits a query to NexLog to retrieve some of the most recent calls for that channel. The MediaRecordEntity includes a GUID which uniquely identifies the call. See NexLog::getRecordsForChannelName for details on how to build a query filter.

4.9. Applying Metadata

NexLog provides the ability to associate user-customized information with each call. The metadata field names and types are entered via the front panel or via Mediaworks Express, and the values of these fields (also called “custom fields”) are available in the MediaRecordEntity object (see Retrieving Call Data).

4.10. Call Breaks

A “call break” can occur when metadata is received by NexLog during a recording. Depending on the metadata “action” the call may be broken into one or more recordings on the NexLog, each recording being associated with its unique metadata. The .NET application shows how metadata can be applied to a recording and when to use a call break.

Three actions are specified, and the NexLog will choose on of them depending on the channel’s type and whether or not a recording is in progress.

The metadata actions are:

Action	Description
NONE	apply the metadata to the current recording
APPLY_BREAK	apply the metadata to the current recording, stop the recording and immediately start another
BREAK_APPLY	top the current recording, immediately start a new recording and apply the metadata to it
CACHE	cache this metadata and apply it the next time a call starts
DELETE	schedule the current recording for deletion
START	Start a new call and apply the metadata. This generally applies to analog audio sources only, although digital sources can be configured to accept this action
STOP	apply the metadata and stop recording

If a recording is in progress then the accepted actions are NONE, STOP, APPLY_BREAK, BREAK_APPLY and DELETE.

If a recording is not in progress then the accepted actions are NONE, START, CACHE and DELETE.

```
// code snippet
return _client.setChannelMetadata(channel,
      callHasMetadataStartAction,
      noCallInProgressAction,
```

```
        callInProgressAction,
        metadata);
```

4.11. Channel Recording Control

A NexLog may be fitted with an analog telephone interface. This kind of interface can be set to record at any time, even if a call is not in progress. Digital channels (E1/T1, SIP Trunk etc) can also be set to act in this way, although this usage is not common and the facility is not provided by the .NET application. Recording can be initiated with the “Record” button.

Either type of channel can be set to stop recording at any time; this action is invoked with the “Stop” button.

```
// code snippet
// Start and stop recording with a 2 second timeout
//
public ServiceReference2.RecordingStatus
startRecording(int channelNumber)
{
   return _client.startRecording(channelNumber, 2);
}
public ServiceReference2.RecordingStatus
   stopRecording(int channelNumber)
{
   return _client.stopRecording(channelNumber, 2);
}
```

4.12. Squashing a Channel

When a channel is put into “squashed” state, it is actively recording but the recording information is replaced with silence. This might be used when accepting credit card information or other confidential information.

```
// snippet
if (true == squash)
{
_client.recordDisable(channelNumber);
}
```

```
else
{
    _client.recordEnable(channelNumber);
}
```

4.13. Supporting Information

Extended WPF Tookit is at http://nuget.org/packages/Extended.Wpf.Toolkit

Your NexLog WSDL is at http://YourLogger/soap.fcgi?wsdl

This page was intentionally left 99.88% blank.

5. INTERFACING TO NEXLOG'S REST API

5.1. Authentication

Authentication (username and password) information is passed via POST form data

POST /client/apps/login/

Form Parameters

- **location** – localhost
- **username** – user's username
- **password** – user's password

Example:

```
POST /client/apps/login/ HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
X-Requested-With: XMLHttpRequest
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Accept: */*

location=localhost&username=Eventide&password=12345
```

5.1.1. Successful Authentication Response

The JSON object contains `sessionKey` which indicates a successful login. The session key does not need to be saved but the cookie information in the Set-Cookie HTTP header does need to be a part of future communications.

Example:

```
HTTP/1.1 200 OK
Cache-Control: no-store, no-cache, must-revalidate
```

```
Set-Cookie: eventide_soap_session=ZP1zlRUwExHcl5Wc; Path=/
Set-Cookie: eventide_soap_user=Eventide; Path=/
Content-type: text/html


{
   "username":"Eventide",
   "license":{
      "MP3":false,
      "SpeechToText":false,
      "hasATCMode":false,
      "hasNAB":true,
      "hasQuarantine":false,
      "hasRapidSOS":false,
      "packagedIncidentExport":false
   },
   "location":"localhost",
   "loginRc":-1,
   "password_expire_days":-1,
   "result":"success",
   "security":{
      "admin":"true",
      "system_security":null,
      "user_groups":{}
   },
   "sessionkey":"iOiw2LJyvggt7Sr8",
   "tos":""
}
```

5.1.2. Failed Authentication Response

A failed login is indicated by populated `error` and `errorDetail` fields

Example:

```
HTTP/1.1 200 OK
Cache-Control: no-store, no-cache, must-revalidate
Content-type: text/html


{
   "loginRc":-1,
```

```
    "password_expire_days":-1,
    "location":"localhost",
    "errorDetail":"Authentication failure",
    "error":"AUTH_FAULT"
}
```

If the session has timed out or an operation is being performed when the login has not been established then the JSON object will look as follows

Example:

```
HTTP/1.1 200 OK
Cache-Control: no-store, no-cache, must-revalidate
Content-type: text/html


{
    "isSSO":"0",
    "errorDetail":"Not logged in",
    "error":"NO_SESSION_FAULT"
}
```

5.2. Retrieving Recording Data

This method queries the system for records.

GET /apps/rest/call.json

Query Parameters

- **type** (*string*) – stackable return type (`json` , `metadata` , `array`)
- **timezone** (*string*) – timezone for the start and end time, default UTC
- **columns** (*string*) – comma separated list of data to return for each record, also uses `call.X` parameters below
- **call.channelid** (*int*) – channel number to return records for
- **call.datastatus** (*char*) – current state of the record (`c` , `p`)
- **call.callguid** (*string*) – unique id of the record
- **call.calldirection** (*string*) – inbound or outbound direction (`i` , `o`)
- **call.duration** (*int*) – length or the record in seconds

- **call.starttime_timestamp** (*timestamp*) – record's start time
- **call.stoptime_timestamp** (*timestamp*) – record's end time
- **call.sourceid** (*int*) – serial number of the record's originating system
- **call.isarchived** (*boolean*) – if the record has been sent to an archive

Example Request:

The following example retrieves the callguid for the call in progress (call.datastatus=p) on channel 1 (call.channelid)

```
GET /client/apps/rest/call.json?
type=json,metadata,array&timezone=America/
New_York&columns=call.channelname,call.callguid&call.channelid=1&ca
ll.datastatus=p HTTP/1.1
Host: 192.168.1.1
Connection: keep-alive
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: eventide_soap_user=Eventide;
eventide_soap_user_192.168.1.1=Eventide;
eventide_soap_session=CxaxD4SjxgESH44J;
eventide_soap_session_192.168.1.1=6xfRiDZmXz43fzN2
```

Example Response:

The returned JSON object contains the callguid in the `rows` parameter

```
HTTP/1.1 200 OK
Content-Type: application/json


{
   "count":1,
   "rows":[
      [
         "Channel One",
         "0XvGC4DKwSeOQv1m"
      ]
   ],
```

```
    "timestamp":1010181746,
    "rownumber":0,
    "offset":0,
    "total":1,
    "columns":[
       "channelname",
       "callguid"
    ]
}
```

5.3. Retrieving Data Based on Custom Fields

This method queries the system for records based on data in a Custom Field. All of the `call.X` parameters above can be used here as well.

GET /apps/rest/call.json

Query Parameters

- **type** (*string*) – stackable return type (`json` , `metadata` , `array`)
- **timezone** (*string*) – timezone for the start and end time, default UTC
- **columns** (*string*) – comma separated list of data to return for each record using `call.X` parameters above and `userdefinedcallmetadata.X` which will vary from system to system
- **userdefinedcallmetadata.CAD_INCIDENT_ID** (*string*) – a custom field with CAD data relating to the calls (make sure Custom Field names are in all caps)

Example Request:

The following example retrieves the start time (call.starttime_timestamp) and duration (call.duration) for a call with Custom Field CAD_INCIDENT_ID equal to QAZWSX

```
GET /client/apps/rest/call.json?
type=json,metadata,array&timezone=America/
New_York&columns=call.starttime_timestamp,call.duration&userdefined
callmetadata.CAD_INCIDENT_ID=QAZWSX HTTP/1.1
Host: 192.168.1.1
Connection: keep-alive
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: */*
```

```
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: eventide_soap_user=Eventide;
eventide_soap_user_192.168.1.1=Eventide;
eventide_soap_session=mZoFvzdFqa6NIe8O;
eventide_soap_session_192.168.1.1=ei2zJufMmiK3OqXr
```

Example Response:

The returned JSON object contains the start time and duration in the `rows` parameter

```
HTTP/1.1 200 OK
Content-Type: application/json


{
   "count": 1,
   "rows": [
      [
        1706591167732,
        301
      ]
   ],
   "timestamp": 1107124117,
   "rownumber": 0,
   "offset": 0,
   "total": 1,
   "columns": [
      "starttime_timestamp",
      "duration"
   ]
}
```

Note

Custom Fields that are not prefixed by "*call.*" or those that show up in the Custom Fields page in the Configuration Manager (not part of the call table) are required to be uppercase when included in the query.

5.4. Associating Metadata to a Record

POST /client/apps/soap/

Form Parameters

- **location** – `localhost` **
- **method** – action to take against record `setCallMetadata` **
- **channelid** – channel number that the record is from
- **callguid** – unique record id to set metadata for **
- **keys** – array of metadata fieldnames to set **
- **values** – array of values corresponding to the keys **

** *required*

Example Request:

The following example sets the `user_id` custom field associated with call GUID `HfE0YV13RO3rVVwv`. The field value is set to `Taylor Smith`.

```
POST /client/apps/soap/ HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
X-Requested-With: XMLHttpRequest
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Accept: */*

location=localhost&method=setCallMetadata&callguid=HfE0YV13RO3rVVwv&
keys=["user_id"]&values=["Taylor%20Smith"]
```

Example Response:

If the operation is successful the server will return

```
HTTP/1.1 200 OK
Content-type: application/json
```

```
{
   "result": true
}
```

5.5. Adding Annotations to a Record

POST /client/apps/soap/

Form Parameters

- **location** – `localhost` **
- **method** – action to take against record `setCallMetadata` **
- **callguid** – unique record id to set metadata for **
- **keys** – array of metadata fieldnames to set **
- **values** – array of values corresponding to the keys **

** *required*

Example Request:

The following example adds to, but does not overwrite, the `annotations` custom field associated with call GUID `HfE0YV13RO3rVVwv`. The annotation value is set to `Agent properly greeted caller`. The annotation time is a UNIX epoch timestamp with milliseconds and a value of `1706629349628` which equates to January 30, 2024 15:42:29.628 UTC. The annotation type is set to `GOOD`. Other annotation types can be seen in the Configuration Manager in Recording → Custom Fields, select `ANNOTATIONS` and click Edit Field.

```
POST /client/apps/soap/ HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
X-Requested-With: XMLHttpRequest
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Accept: */*

location=localhost&method=setCallMetadata&callguid=HfE0YV13RO3rVVwv&
keys='["%2BANNOTATIONS","ANNOTATION_TIME","ANNOTATION_TYPE"]'&
values='["Agent properly greeted caller","1706629349628","GOOD"]'
```

Example Response:

If the operation is successful the server will return

```
HTTP/1.1 200 OK
Content-type: application/json


{
   "result": true
}
```

5.6. Example Bash Script

This example script queries the system for a recording in progress on channel 1. It then uses the `callguid` and `channelid` to set the `User ID` field to `Taylor Smith` for that record. Lastly, it uses the `callguid` and `channelid` to add an annotation to the `annotations` custom field and sets the annotation type to GOOD which will display in MediaWorks DX as a green circle with a checkmark.

> **Note**
>
> The key used for adding annotations is `+ANNOTATIONS` but the example below uses the URL escaped `%2B` instead of `+` to make sure that wget doesn't treat the `+` as an escaped space.

```
#!/bin/bash
NLHOST=192.168.22.98
USERNAME=Eventide
PASSWORD=12345
CHANNELID=1
LOGFILE=logfile.txt
COOKIES=cookies.txt

# Authenticate
wget -a ${LOGFILE} --save-cookies ${COOKIES} --keep-session-cookies --post-
data location=localhost\&username=${USERNAME}\&password=${PASSWORD} -O
auth.json http://${NLHOST}/client/apps/login/
```

```
# Get the guid of the call currently in progress
wget -a ${LOGFILE} --load-cookies ${COOKIES} --keep-session-cookies -O
query.json "http://${NLHOST}/client/apps/rest/call.json?
type=json%2Cmetadata%2Carray&timezone=America%2FNew_York&columns=call.callgu
id&call.channelid=${CHANNELID}&call.datastatus=p"

# extract call guid from JSON
queryraw=`cat query.json`
CALLGUID=`echo "{"${queryraw#*{} | jq -r '.rows[0][]'`

# use channelid and callguid to set user_id
wget -a ${LOGFILE} --load-cookies ${COOKIES} --keep-session-cookies --post-
data location=localhost\&method=setCallMetadata\&channelid=${CHANNELID}
\&callguid=${CALLGUID}\&keys='["user_id"]'\&values='["Taylor%20Smith"]' -O
setMetadata.json http://${NLHOST}/client/apps/soap/

# use channelid and callguid to add an annotation at annotation_time and
pinned with annotation_type GOOD
wget -a ${LOGFILE} --load-cookies ${COOKIES} --keep-session-cookies --post-
data location=localhost\&method=setCallMetadata\&channelid=${CHANNELID}
\&callguid=${CALLGUID}
\&keys='["%2BANNOTATIONS","ANNOTATION_TIME","ANNOTATION_TYPE"]'\&values='["Agent
properly greeted caller","1706629349628","GOOD"]' -O setMetadata2.json
http://${NLHOST}/client/apps/soap/
```

6. NEXLOG GENERIC CAD API

6.1. Overview

The NexLog Generic CAD API provides a simple mechanism to allow CAD systems to inject data about call records into the NexLog database, where they are stored, archived, retrieved, and displayed along with the call records and the native metadata stored with the call. The Generic CAD API allows arbitrary additional metadata fields including call ids and incident id's to be provided from the CAD system to the recorder. Users of the NexLog system can then choose to display and/or search based on these metadata values associated with the call records.

The NexLog Recording system has a facility for 'Metadata Feeds'. A Metadata Feed is a configurable TCP, UDP, or Serial Input, configured to expect data in a given data format. The Metadata Feed framework allows the format of a given metadata feed to be programmatically defined in a configuration file on the recorder and can support a myriad of different input formats. The Generic CAD API is implemented as a specific realization of this more generalized feature. To use the NexLog Generic CAD API on a given recorder, a License for 'Generic CAD API' must be purchased from Eventide for the specific recorder. By purchasing this license, the end user will receive a license key to activate the Metadata Feeds recorder feature, the specific configuration file to load on the recorder to support the Generic CAD API data format, and the legal permission to utilize the feature on the recorder. The Generic CAD API licensed feature must be purchased and installed on a given recorder prior to using the Generic CAD API.

6.2. Initial Setup

To enable and configure the Generic CAD API on a given recorder, first log into the recorder's Configuration Manager via a web browser as a user account with administrative access to the recorder. First, navigate to System->License Keys and add the license key for 'Metadata Feeds' if this has not already been installed. Next, under System->Configuration Files, find the Metadata Feeds configuration file and paste in the 'Generic CAD API' Configuration file provided when the feature was purchased. This will enable the feature with the default settings. See sections below about modifying default settings if required for your usage.

6.3. Transport Mechanism

The Generic CAD API allows ASCII formatted commands to be sent to the recorder in predefined formats to accomplish various tasks related to recorder metadata. By default, the Generic CAD API will expect these commands to be sent to the recorder as UDP on port 5000, but both of these settings are configurable. When using UDP, the recorder will expect to receive one full ASCII command per UDP datagram. To alter which port the recorder will listen on for commands, look near the top of the configuration file for a line which says

```
InputPort = 5000
```

This can be changed to any other available port that you wish the recorder to listen on for Generic CAD API Messages. Instead of listening on a UDP port, it is also possible to configure the Generic CAD API to listen on a TCP Port. To configure TCP Mode, change the line near the top of the configuration file from:

```
InputType = UDP
```

To:

```
InputType = TCP
```

The recorder will now listen on the configured port number as a TCP port instead of UDP. Unlike UDP, which is stateful, and each inbound UDP message can come from a different source, TCP is a stateful, connection based protocol. The Generic CAD API when configured in TCP mode will expect a single TCP connection to be made from the CAD system to its configured TCP port, and will expect ASCII commands to be sent to this TCP port. It is the CAD system's responsibility to maintain the TCP connection, detecting when it is no longer connected, and reconnecting as needed. Multiple simultaneous TCP connections are not supported. Connecting an additional TCP connection will automatically disconnect any established connection.

In addition to TCP and UDP, the Generic CAD API can be configured to listen on a serial port instead of on a TCP or UDP port. To configure this mode, change the line near the top of the configuration file that says:

```
InputType = UDP
```

To:

```
InputType = Serial
```

And add in the additional required configuration lines immediately below. For example:

```
SerialType = n81
SerialBaud = 9600
SerialDevice = /dev/ttyS0
```

The SerialType configuration setting defined the parity, number of data bits, and number of stop bits for the serial connection, for example, N81, or E72. The SerialBaud configuration setting defines the baud rate to be used on the serial connection. Finally, the SerialDevice configuration setting tells the recorder which serial port to connect to the Generic CAD API for receiving commands. A standard NexLog has two serial ports, which are referenced as /dev/ttyS0 and /dev/ttyS1. If additional serial port add-in cards have been purchased and installed into the NexLog, then they can be accessed as /dev/ttyS2, /dev/ttyS3, etc.

6.4. Verification of Data Received

Generic CAD API can receive data over TCP, UDP, or Serial, but regardless of the transport mechanism used, the format of the data conveyed is identical. The Data format is ASCII text, so the commands are human readable over the transport layer. Each command is sent followed by a newline (‘n’) character, so each command is a separate line. Commands will have varying number of parameters, and the meaning of those parameters differs depending on the command, but the command format is always the same. The example below is a 4 parameter command; though various numbers of parameters are possible depending on the command.

```
<Command:Param1:Param2:Param3:Param4>\n
```

Commands open with a left bracket character (‘<’), and close with a right bracket character (‘>’). The command itself and all its parameters are separated by colon (‘:’) characters. If a parameter needs to contain a right bracket (‘>’) or a colon (‘:) character, these are reserved characters and need to be

escaped as to not interfere with processing. To insert these characters into parameters escape them as \COLON and \RIGHTBRACKET (a backslash followed by the actual name of the character all in uppercase).

Upon receiving a command, the API will attempt to parse it. If the format matches a valid command, the API will respond with the ASCII text

```
OK\n
```

If the command is not successfully parsed, no response will be sent. Note that the OK response verifies only that the command was successfully parsed and passed on to the rest of the system. It does not guarantee that the command actually had its desired effect. For example, if a command tells the system to apply data to a given metadata field, and that metadata field does not exist, the command will parse successfully and return OK, but its actual result will be to raise an alert on the recorder warning about the missing metadata field.

6.5. API Command Data Format

In the NexLog Configuration Manager, you can verify what data has been received by the Generic CAD API by going to Utilities -> Metadata Feeds and selecting Generic_CAD_API and clicking View Feed. From this screen, you can View Input Logs for the Generic_CAD_API metadata feed which will show all recent data received by the configured input port (whether UDP, TCP, or Serial). This can be useful to verify that the data is being properly sent and received. It is also possible from this screen to View Processing Logs, which shows the received messages color coded and describes how they are being extracted and processed by the Generic CAD API. A command is displayed in the Processing Logs once it is successfully identified and parsed by the API, whereas the Input Logs show every character received, regardless of proper formatting, etc.

6.6. Physical Channel Commands

There will be multiple groups of commands described in this document. The first group to be handled are the commands that reference a physical channel on the recorder. Every NexLog recorder has a certain number of channels installed and configured, up to a maximum of 255 channels. For example, inserting a 24 channel Analog board into the recorder will add 24 channels to the recorder, each of which is hardware defined by two pins on the analog card; when audio is sent to those pins, they record on the hard coded corresponding recorder channel. In addition to Analog, Digital PBX, and T1/E1

tapping boards, channels can belong to 'Virtual' boards that are installed and licensed, such as ScreenAgent channels, and VoIP channels. Though there are not physical pins and wires in these configurations, each channel number is still hard coded and defined by a single input, for example a particular IP and Port, or a particular PC Workstation. Physical channel commands allow metadata to be assigned to the current call recording on a known physical channel. Physical channels are the simplest available commands for attaching metadata to recordings and are most useful when the physical recording channel has a one to one correspondence with the CAD system. For example, if individual channels on the recorder are recording audio feeds from individual CAD positions and the CAD system sends per-position metadata, or when individual recorder channels are recording audio feeds from individual Trunks, and the CAD system sends per-trunk metadata. When this correspondence does not exist due to the way the recorder and CAD system are configured and connecting, physical channel commands cannot be used, and other types of command described later in this document should be used instead.

Physical Channel Commands have the following format:

```
<Command:ChannelNumber:Field1:Value1:Field2:Value2:Field3:Value3: .....Field8:Valu
```

Though they can have less than 8 Field/Value fields if less metadata needs to be attached to the call.

The Following Physical Channel Commands are available:

ATTACH: Attach the given metadata to the current recording call on the physical channel. If No call is currently recording, attach it to the most recent call to complete on the channel. Note that if a call record already has a value in a given field that is included in the command, that field will be overwritten with the new value.

ATTACH_OR_CACHE: Attach the given metadata to the current recording call on the physical channel. If No call is currently recording, attach it to next call to start on the channel

BREAK_ATTACH: If a call was started using a CAD API START command and is recording then it will trigger a call break, start a new call, and attach metadata to this new call. If a call was started without using the CAD API START command (example: triggered by calls based on the different detect type of the channel) and is recording, then it will not trigger a call break but attach the metadata to the current call. If no call is currently recording on the channel then BREAK_ATTACH will start recording a new call and attach the specified metadata to this new call.

BREAK_ATTACH_OR_CACHE: If no call is currently recording on the channel, it will attach the metadata to the next call to start on the channel, otherwise it will perform the same action as BREAK_ATTACH.

Under normal circumstances, the recorder will start and stop recordings based on its own internally configured criteria, for example, VOX levels and hold times, or Tip/Ring voltages, and the CAD API will be used only to attach metadata to existing calls (and possibly break a call into multiple pieces so different metadata can be attached to different segments). It is possible, however, with certain recorder configurations, for the Generic CAD API to be used to control the starting and stopping of recording on the physical channel. These commands are only available for physical channels. Note that in for these commands to work on a given channel, the channel itself must be configured to allow scripted control and not to use the available internal mechanisms to control recording. The following additional call control commands are available for physical channels:

START: If No call is in progress on the physical channel, begin recording, and attach the metadata given in the command to the new call. If a call is already in progress, attach the metadata to it

START_OR_BREAK: If No call is in progress on the physical channel, being recoding and attach the metadata given in the command to the new call. If a call is already in progress, stop it, start a new call, and attach the metadata to it

STOP: If no call is in progress, attach the metadata to the most recent call on the channel, otherwise stop recording immediately and attach the given metadata to the recently stopped call.

In addition to attaching metadata to call records and starting, stopping call records, there is one additional command which creates a standalone call record with no audio to store an event. This command could be used, for example, to insert a received text message into the recorder, or to insert a note about the console position going off line, or a new agent logging into it, etc.

EVENT: Regardless of whether or not a call is present on the channel, create a call record with no duration or audio and place it at the current time on the physical channel. This command is especially useful with the ANNOTATION metadata field to create standalone annotation pins on the timeline. For example, an Event could be given an annotation text, ChannelName, Caller_ID, and CallType of ‘TEXT’ for a text message arriving at the CAD position. Unlike the other physical channel messages, this one is useful even if there is no one-to-one physical channel mapping, the messages can be hard coded to a specific physical channel such as ‘1’, and then use the ChannelName metadata field to make sure it appears where you want it to. Since there is no actual audio, the physical channel is not important in most cases.

While the ChannelNumber given in the physical channel commands can be the actual physical channel number (for example 12 for the 12th channel on the recorder), using this mechanism requires the CAD system to know which position or trunk, etc., is wired to which physical channel number on the recorder, which would require the CAD side of the integration to be altered per-site to put in the matching recorder channel numbers for the position, trunk, etc. To work around this issue, the Generic

CAD API allows a mapping table to be configured on the recorder from a given value to a channel. For example, the CAD system could send as the channel 'POSITION 1', or '555-1212' or '911 Trunk 12', and when the Generic CAD API configuration file is loaded onto the recorder, it can be altered to contain the mappings from these tags to actual physical channel numbers. Again, it is important to note, this can only be done when there is a one-to-one correspondence between the tag and a physical channel, e.g. 'POSITION 1' is Channel 43. If CAD commands are being sent for a given position, but the recorder is recording Trunks, there is not a defined relationship that 'Position 1' is Channel X, calls for position 1 could record on any channel, and in this configuration, physical channel commands cannot be used, and more advanced commands described in sections below must be used instead.

To configure a mapping from CAD supplied tags to physical channel numbers, edit the Generic CAD API configuration file loaded onto the recorder. Look for the section:

```
[ChannelMap]
MapType = Lookup
Map = 1 <- Example1
Map1 = 2 <- Example2
```

The supplied example mappings would map the tag Example1 to channel 1 and Example2 to channel 2. Regardless of the ChannelMap, if an integer from 1-255 is sent as the channel number in a Generic CAD API command, then the corresponding physical channel ID will automatically be used. The mapping only needs to be supplied if alternative tags are to be used. The two example entries can be modified to suit your purposes, and additional Map entries can be added, e.g. Map2 = 6<-Position 1 would add a new mapping from the tag 'Position 1' to channel 6. Note that each Map value should be unique and in ascending order without any gaps, Map, Map1, Map2, Map3, etc. but otherwise the number appended to the word map have no semantic meaning. The format is MapX = <channel ID> <- <Tag Name>

After the command and the physical channel number come between zero and eight Field/Value pairs of metadata to attach to the call. (Though zero would only be useful in the case of a START or STOP command, as attaching no metadata to a call and taking no action would not do anything, and not be a command worth sending). The field should be the name of a recorder metadata field, and the value is the value you wish to place in that metadata field for the call record referenced by the command. Generic CAD API commands can reference any metadata field that has been added to the recorder via the NexLog Configuration Manager, including default fields that are shipped with every recorder, such as Caller_ID, CallType, and DTMF. In addition to filling in existing metadata fields, Generic CAD API based integrations can utilize arbitrarily named additional metadata fields defined by the CAD integrator / API user. It is recommended that these metadata fields be added to the NexLog recorder at the time the Generic CAD API configuration file is loaded onto the recorder. If the CAD system

attempts to apply metadata to a field that has not been created on the recorder, the metadata will not be attached, and an alert will be raised on the recorder about the mismatch between configured fields and the fields expected by the integration.

The following built-in and default metadata fields have special meaning:

CHANNELNAME: The name of each physical channel can be configured in the NexLog Configuration Manager. Each call on that physical channel will default to the CHANNELNAME of the physical channel it was recorded on. However, using the Generic CAD API, the CHANNELNAME can be modified from this default to provide an alternate name. For example, it might be desired that while the physical connection is 'Position 1', to set the CHANNELNAME for each call to 'Position 1 - Admin' or 'Position 1 - Emergency' if the position takes both administrative and 911 calls. The CHANNELNAME is the primary way that calls are searched by users of the NexLog system and also the names available in the browse tree in the Browse tab in MediaWorks DX.

CALLER_ID: This field is used, not only for actual caller ID, but is generally the primary 'Calling Party' field on NexLog and is a default field in MediaWorks DX and the Front Panel.

DTMF: This field is used not only for parsed DTMF tones, but is generally the primary 'Called Party' field on NexLog and is a default field in MediaWorks DX and the Front Panel.

CALLTYPE: The CALLTYPE of each physical channel can be configured in the NexLog Configuration Manager. Each call on that physical channel will default to the CALLTYPE of the physical channel it was recorded on. However, using the Generic CAD API, the CALLTYPE can be modified from this default to provide an alternative CALLTYPE. For example, a channel might be configured to mark the call type as 'PHONE' or 'POSITION', but it may be desirable to mark calls as being of type 'FIRE' or type 'EMERGENCY'. CALLTYPE is configured to display an icon (if available) as opposed to the actual text in MediaWorks DX. The icon associated with a given text string is configured in the NexLog Configuration Manager under Recording->Custom Fields. While any metadata field can be configured to use icons, CALLTYPE is special in that not only do its icons appear in the CALLTYPE column in the call grid, but there is an option in MediaWorks DX to turn on CallType icons in the timeline, where the configured icon is shown on top of the call record in the timeline as well.

ANNOTATIONS: Annotations are snippets of text that occur at a specific time during the call. Unlike other metadata fields, where a subsequent update will overwrite old values, a single call record can have multiple annotations. MediaWorks DX will show annotations as pins on calls where the text they contain can be seen on mouse-over. The mouse-over effect will also happen automatically during playback to create an annotation on the call, enter the text you wish to annotate onto the call into this field

ANNOTATION_TYPE: Only valid if the command also has ANNOTATIONS. If ANNOTATIONS is present, and ANNOTATION_TYPE is not, ANNOTATIONS default to type SYSTEM which means they display as a gear icon on the MediaWorks DX timeline, and are not user editable/deletable. If ANNOTATION_TYPE is set to USER, then they appear as a pin in the timeline and are editable / deletable by users. These are currently the only valid annotation types, though more may be added in future NexLog firmware release.

ANNOTATION_TIME: Only valid if the command also has ANNOTATIONS. If ANNOTATIONS is present and ANNOTATIONS_TIME is not, the ANNOTATION time defaults to the current recorder time when the command was received. If present ANNOTATION_TIME should contain a UTC timestamp, in the format of number of milliseconds since UNIX Epoch.

LOCATION: This metadata field is only available if a GeoLocation license is installed on the recorder. The LOCATION metadata field must be populated with data in a specific format. The valid format is (latitude,longitude) including the parenthesis. The LOCATION metadata field represents the GPS coordinates from which the call originates. If present, and the necessary licenses are loaded on the recorder, this allow the recorder to display the call as a pin on a map, and also allows searching for calls based on location (The user can draw a shape on the map to find calls that occurred inside of it).

In addition to these fields, any additional desired metadata fields can be created and used. Don't forget to escape '>' and ':' characters in values as described above. Up to eight key/value pairs can be provided, or they can be left out. If left out, leave them blank, but still provide the correct number of colons in the command. Note also that while most fields are created in the Configuration Manager as Text fields, meaning they can contain any ASCII (or UTF8) data, it is possible to create integer only fields, where if non integer data is applied to them, the insert will fail and the recorder will raise an alert.

Examples:

```
<ATTACH:43:CALLTYPE:FIRE:CALLER_ID:(201)-555-1212:LOCATION:
(40.8529,-74.0421)::::::::::>\n
```

Would attach metadata to the current call on channel 43 (or most recent call on 43 if one is not currently in progress.) CallType, Caller_ID, and Location fields are set. Note the extra colons for unused metadata fields:

```
<ATTACH_OR_CACHE:Position_1:CALLER_ID:(201)-551-1212:INCIDENT_ID:
160628-43:::::::::::>\n
```

Would attach metadata to the current call on the channel mapped to the tag ‘Position_1’ in the Generic CAD API configuration file, or cache it for the next call to start on the channel if no call is currently in progress. The metadata attached will be a Caller_ID, and an INCIDENT_ID field from the CAD system which is not a standard NexLog field, but has been added to the recorder using the Configuration Manager.

One additional Physical Channel Based Command is available:

ATTACH_AT_TIME: Attaches the metadata to the call that was in progress at a given time on the given physical channel. The first parameter is the physical channel ID or a tag that is configured to map to a physical channel id in the Generic CAD API configuration file on the recorder. The second parameter is the time, in seconds since UNIX Epoch (UTC). After that are up to eight field/value pairs to set.

```
<ATTACH_AT_TIME:22:1467909534:INCIDENT_ID:2016-444:::::::::::::>
```

Would set the INCIDENT_ID metadata field to 2016-444 for the call that was in progress at 1467909534 UNIX Epoch time (which is July 7, 2016 at 16:38:54 UTC) on physical channel 22. If no such call exists, the metadata is attached to the most recent call to end before the given time on the physical channel.

6.7. Non-Physical Channel Commands

Non-physical channel commands differ from physical channel commands in that they do not include a physical channel number (or physical channel mapping tag) and cannot perform actions on the call (such as call breaks, starting, or stopping calls), only attach metadata. They work, not by specifying ‘The Current Call on Channel X’, to act upon, but by specifying some condition that must be met by a call in order for the metadata to be attached for it. For example, a command could say “Attach this metadata to all calls that have Caller_ID=(201)-555-1212.” At the point the command is run, any call that currently meets the given criteria will have the metadata attached. Note that unlike physical channel commands, a non-physical channel command can act on more than one call record if more than one call record meets the given criteria. Non-physical channel commands will only affect calls that are no more than 24 hours old. Any call less than 24 hours old that matches the given criteria will be affected.

Non-physical channel commands can be used to attach data to call records based on the metadata they already have attached to them, via other interfaces, including ANI/ALI, SIP Signaling, MF Tones, etc. The call must already have the matching values at the point in time when the command is sent. There are two variants of the non-physical channel command:

ATTACH_ON_MATCH: Attaches the metadata to all calls from the past 24 hours that match the given criteria

ATTACH_ON_MATCH_1: Attaches the metadata to only the call that matches the given criteria with the most recent start time (must be within 24 hours).

The first two parameters for a non-physical channel command are the metadata field and value to match on, the rest (up to eight) are the metadata to attach, e.g.:

```
<ATTACH_ON_MATCH_1:CALLER_ID:2015551212:INCIDENT_ID:2016-4432::::::::::::>
```

Would set the INCIDENT_ID metadata field to 2016-4432 to the most recent call on the recorder that has the CALLER_ID metadata field set to 2015551212. The value to match must be exact, and is case sensitive.

One additional command is available:

ATTACH_ON_MATCH_AT_TIME: Attaches the metadata to the call that was in progress at a given time that matches the given field/value. The first parameter is the time, in seconds since Unix Epoch (UTC). The second and third parameters are the field and value to match, and after that are up to 8 field/value pairs to set. This command is most useful for attaching to a call on a given ChannelName at a given time regardless of the physical channel. For example:

```
<ATTACH_ON_MATCH_AT_TIME:1467909534:CHANNELNAME:Position12:INCIDENT_ID:
2016-444::::::::::::>
```

Would set the INCIDENT_ID metadata field to 2016-444 for the call record that was in progress at 1467909534 UNIX Epoch time (which is July 7, 2016 at 16:38:54 UTC) which also had the ChannelName set to ‘Position12’, regardless of what physical channel the call was recorded on. If no such call exists, the metadata will be attached to the most recent call to complete before the given time that meets the given criteria.

6.8. Text Call Commands

Text Call commands can be used to attach text messages to text call records. The following format applies for both Physical and Non-Physical Channels:

```
<command: ChannelNumber: Sender's Name: Text Body>
```

ATTACH_OR_CACHE_TEXT: Attach the given text body to the current recording text call on the physical channel. If no text call is currently recording, attach it to the next text call to start on the channel. This option is intended for use with analog channels. Use the START commands to start recording calls on analog channel to create a call for this text record. Either the STOP command or VOX timeout can be used to end text calls for analog.

ATTACH_OR_START_TEXT: Attach the given text body to the current recording text call record on a non-physical channel. If no text call is currently recording, start a text call on the given channel and attach it. This command is intended for use with RTP channels. For RTP, there is no stop command and the RTT Timeout set for the channel is what will be used instead.

For example, for analog channels:

```
<START:42::::::::::::::::::>
<ATTACH_OR_CACHE_TEXT:42:A:What are you doing now?>
<ATTACH_OR_CACHE_TEXT:42:B:I'm watching TV.>
<STOP:42::::::::::::::::::>
```

And for RTP:

```
<ATTACH_OR_START_TEXT:57:A:What are you doing now?>
<ATTACH_OR_START_TEXT:57:B:I'm watching TV.>
```

6.9. Agent Login/Workstation Tagging Commands

Agent_(Login/Logout/Info) Commands can be used to log an agent into the Client Activity Table to enable workstation tagging on specified channels. Each of these actions have the following format:

```
<AgentAction:AgentId:ChannelNo:ClientAddress:Field1:Value1:Name2:Value2:…
Field8:Value8>
```

AGENT_LOGIN: Log the Agent into the Client Activity Table and attach the given metadata to the specified channel.

AGENT_LOGOUT: Log the Agent out of the Client Activity Table.

AGENT_INFO: Change the metadata to be attached to the calls that start on the specified channel for an agent already logged into the Client Activity Table.

This page was intentionally left 99.88% blank.

7. REPORTING PROBLEMS

It is Eventide's policy to work directly with dealers. Your dealer must report your problem to Eventide with the following information in order to process the service/support request:

- Serial number(s) of the affected recorder(s).
- Software versions for the recorder(s).
- Severity of the issue, including a detailed description.
- Contact information (phone and email) for the dealer and on-site technician.

To contact Eventide customer service for support, call 201-641-1200 Option 6 followed by Option 2 (Communications/Recorders Division) or email service@eventidecommunications.com.