



Dante Application Library User Guide

For Windows and macOS



Document version: 1.8

Document name: AUD-MAN-DAL_User_Guide-v1.8

Published: Thursday, March 9, 2023



Feedback: if you would like to suggest improvements to this information, please feel free to email us at documentation@audinate.com.



Copyright

© 2023 Audinate Pty Ltd. All Rights Reserved.

Audinate®, the Audinate logo and Dante® are registered trademarks of Audinate Pty Ltd.

All other trademarks are the property of their respective owners.

Audinate products are protected by one or more of US Patents 7747725, 8005939, 7978696, 8171152, European Patent 2255541, Chinese Patent ZL200780026677.0, and other patents pending or issued.

See www.audinate.com/patents.

Legal Notice and Disclaimer

Audinate retains ownership of all intellectual property in this document.

The information and materials presented in this document are provided as an information source only. While effort has been made to ensure the accuracy and completeness of the information, no guarantee is given nor responsibility taken by Audinate for errors or omissions in the data.

Audinate is not liable for any loss or damage that may be suffered or incurred in any way as a result of acting on information in this document. The information is provided solely on the basis that readers will be responsible for making their own assessment, and are advised to verify all relevant representation, statements and information with their own professional advisers.

Software Licensing Notice

Audinate distributes products which are covered by Audinate license agreements and third-party license agreements.

For further information and to access copies of each of these licenses, please visit our website:

www.audinate.com/software-licensing-notice

Contacts

Audinate Pty Ltd

Level 7, 64 Kippax Street
Surry Hills NSW 2010
Australia
Tel. +61 2 8090 1000
info@audinate.com
www.audinate.com

European Office

Audinate Ltd
Future Business Centre
Kings Hedges Rd
Cambridge CB4 2HY
United Kingdom
Tel. +44 (0) 1273 921695

Audinate Inc

1200 NW Naito Parkway
Suite 630
Portland, OR 97209
USA
Tel: +1.503.224.2998
Fax. +1.503.360.1155

Asia Pacific Office

Audinate Limited
Suite 1106-08, 11/F Tai Yau Building
No 181 Johnston Road
Wanchai, Hong Kong
澳迪耐特有限公司
香港灣仔莊士敦道181號
大有大廈11樓1106-8室
Tel. +(852)-3588 0030
+(852)-3588 0031
Fax. +(852)-2975 8042

Contents

Copyright	1
About Audinate	5
About Dante	5
Revision History	6
Overview	7
Features	7
The DAL API	8
Getting Started	9
API Access Token	9
Example DAL Applications	10
Preparation and Build	10
Windows Package	10
Mac Package	10
C++ Examples	11
DalLoopbackTest	12
DalWavePlay	12
DalWaveRecord	12
DalConnectionsTest	13
Windows C# Examples	14
Dal Managed Library	14
Dal Loopback	15
Dal Wave Player	16
Dal Wave Recorder	17
macOS Objective-C Examples	18
Dal Loopback App	18
Adding DAL to Your Application	19
Preparation	19
Compiling	19
Linking	19
Application Packaging and Installation	19
Running	20
Network Interface Selection	20
Activation	20
Using DAL	22
Instance Management	22
Child Processes	22
Activation	23
Audio Transfer	24

Connection Management	24
Connections API vs Embedded Dante API (eDAPI)	25
Operational Overview	25
Available Channels	26
Connections, Persistence and Cleaning up Subscriptions	26
Events	27
Specific Behaviours	28
Server Socket Management	30
Firewall Management (Windows)	31
Command Line Activator	31
Appendices	32
Building and Running the DAL Examples in Xcode	32
Requirements	32
Instructions	32
Notarize the DalLoopback Example Application on macOS	34
Prerequisites	34
Audinate Signed Binaries in the DAL Package	34
Instructions	34
Open the DalUIExamples project file	34
Archive the DalLoopback Application	35
Notarize the DalLoopback Application	37
Load Disallowed DAL Library by System Policy on macOS	38
Library Disallowed Error	38
Reason for the Error	39
Authenticating DAL Binaries on Windows	40
Glossary	42
Index	44

About Audinate

Audinate® is the leading provider of professional AV networking technologies globally. Audinate's Dante platform distributes digital audio and video signals over computer networks, and is designed to bring the benefits of IT networking to the professional AV industry. AV-over-IP (AVoIP) using Dante-enabled products ensures interoperability between AV devices and allows end users to enjoy high quality, flexible solutions – typically with a lower total cost of ownership.

About Dante

Dante is the de facto standard digital media networking solution, using standard IP infrastructure to network devices, and making interoperability easy and reliable. It distributes uncompressed, multi-channel digital media via standard Ethernet networks, with near-zero latency and perfect synchronization.

It's the most economical, versatile, and easy-to-use media networking solution, and is scalable from simple installations to large-capacity networks running thousands of channels. Dante can replace multiple analog or multicore cables with a single affordable Ethernet cable to transmit high-quality multi-channel media safely and reliably. With Dante software, the network can be easily expanded and reconfigured with just a few mouse clicks. Dante technology powers products available from hundreds of partners around the world.

For more information, please visit the Audinate website at www.audinate.com.

Revision History

Version	Date	Change
1.8	10th March 2023	<ul style="list-style-type: none">Updated for DAL 1.2.2Updated Load Disallowed DAL Library by System Policy on macOS
1.7	3rd November 2022	Updated for DAL 1.2.1
1.6	15th June 2022	<ul style="list-style-type: none">Updated for DAL 1.2.0Added Load Disallowed DAL Library by System Policy on macOS to Appendices
1.5	6th December 2021	Fixed broken link
1.4	22nd November 2021	Minor text edit to flows line in Features
1.3	7th October 2021	Note added to Windows C# Examples regarding garbage collection
1.2	27th September 2021	Updated for DAL release 1.1
1.1	4th June 2019	<ul style="list-style-type: none">DalLoopbackTest: Now prompts for interfaceDalConnectionsTest: ARCP port changed to ArcpLocal portConnection Management: Note added re. ARCP port numberAvailable Channels: New content
1.0	2nd May 2019	Promoted to v1.0 (no content changes from v0.7)

Overview

Dante Application Library (DAL) is a private, self-contained Dante instance, packaged in library form to allow integration into a host application on Windows or MacOS operating systems.

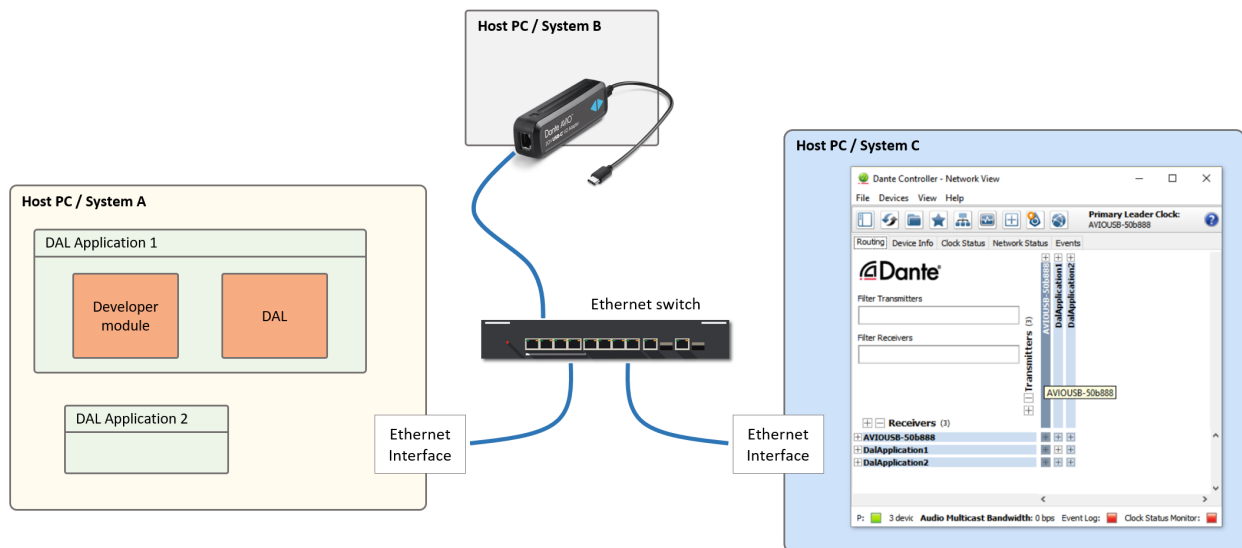


Figure 1 - DAL System Example

The figure above is an illustrative example of how one or several DAL host applications may fit into a Dante network. Any application can be Dante-enabled by including and running DAL. This means the application will appear as a Dante device on the network, and will support many standard Dante features, such as:

- Discoverable and controllable by Dante Controller (and any other controller)
- Can subscribe to audio flows from other Dante devices
- Other Dante devices can subscribe to the application's audio flows

The Developer module shown in the diagram represents the software components that form the non-DAL parts of the application, and the authors of such code are the intended audience of this manual. In relation to DAL, the main responsibilities for the Developer module are:

- Configure, start and stop DAL. One of the main outcomes of DAL configuration is to determine the properties of the resulting Dante device that will appear on the network (number of receive or transmit channels, default device name, sample rate, etc).
- Provide code for the transfer of audio between the DAL channel buffers and the application-specific source/target of the audio

Features

DAL's key features are:

- Co-existence: DAL instances can run concurrently with other Dante instances on the same PC, e.g. Audinate's Dante Virtual Soundcard (DVS) or Dante Via software applications

- Lifecycle management: The DAL instance's lifecycle is controlled by the application
- Non-interference: DAL instances do not interfere with other DAL instances running in other applications on the same PC
- Private audio interface: Applications use a private API to access DAL audio. DAL does not use an audio soundcard interface that is available to all applications on the PC (unlike DVS)
- Connection Management: DAL includes a simple API for local audio routing and management
- Flexible time sources: DAL can obtain network time via networked time protocol (PTP) or from incoming audio packets

DAL supports the following network configuration:

Feature	Supported
Max network audio channels	64 input, 64 output; configurable
Max unicast audio flows	32 receive, 32 transmit; configurable
Network sample rate/s	44.1, 48, 88.2, 96 kHz
Sample rate pull-ups	No
Network encoding	PCM 16/24/32-bit or custom encoding; configurable
Network latency	4ms-10ms; configurable
Time Source	PTP or received audio; configurable
Can be clock leader	No
Dante redundancy	No

The DAL API

The DAL API has three parts:

- The **Instance API** controls the creation and management of a Dante instance. Applications specify the desired configuration for a DAL instance, and control when the instance starts and stops.
- The **Audio API** provides a direct API for getting audio samples to and from a DAL instance. Application call-backs fire when audio is available for sending and/or receiving.
- The **Connections API** provides an optional, simple interface for creating audio connections between the DAL instance and other Dante devices in the network. End users of your application may also control audio connections via Dante Controller. With this API, applications can specify desired connections; the Connections engine will create, maintain and clean up these connections on behalf of the host application. The Connections API may be subject to further enhancement in future DAL updates in response to Developer or end-user needs.



Note: The DAL API makes use of C++11 features. Earlier versions of C++ are not supported.

Getting Started

There are a number of items you will need to begin using DAL:

1. The DanteApplicationLibrary package, e.g. `DanteApplicationLibrary_cp-cpp11-[x.x.x.x]._windows.zip` for Windows and `DanteApplicationLibrary_cp-cpp11-[x.x.x.x]._macos.tgz` for macOS. This package contains the DAL libraries, support binaries and example source code to get DAL up and running.
2. An access token, supplied as a separate C source file - `access_token.c`. This is required to connect to the DAL library. Note that access token comes in two flavours - time-limited evaluation, or permanent once activated.
3. The Dante-Activator package, e.g. `Dante-Activator_[x.x.x.x]._windows.zip` for Windows and `dante_activator-[x.x.x.x]._x86_64_Linux.tgz` for macOS. This package contains the utility that is needed to activate DAL applications.
4. If using an activation (non-evaluation) token, an entitlement ID will be needed for activating DAL.

After unpacking the DanteApplicationLibrary package you will see the following folders:

- `bin` - Contains the DAL shared library and binary files used during DAL runtime
- `include` - All header files required to include in your application
- `lib` - Debug and Release builds of the libraries used to include DAL in your application
- Various example directories containing source code and project / build files, for example DAL applications. See [Example DAL Applications](#) for more details on the provided examples.

Dante devices are designed to run on wired Ethernet networks only. When running your DAL software you will need to select a valid wired network interface.

API Access Token

Audinate will have provided you with an additional source file outside the DanteApplicationLibrary package: `access_token.c`.

This source file contains a cryptographic string unique to your company, which is used to provide proof of access to the Dante Application Library.

There are two types of access tokens:

- Evaluation token: Has an expiry time built-in, but an application using this token does not need to be activated. The date of expiry is shown in the source file. *This should not be used for final release applications.*
- Activation token: Does not have an expiry, but applications using this token will require activation.

This file should be included in your application, and the included cryptographic string passed as an argument on initial connection with DAL API. **This file must not be shared with anyone outside your company.**

Example DAL Applications

Preparation and Build

There are a number of different example sets provided in the DanteApplicationLibrary package, as summarised below.

Windows Package

Example Set (reference name used in this document)	Description	Applications Included	Source Directory
Windows-CPP	C++ command line apps. Visual Studio 2015 solution provided.	DalLoopbackTest, DalConnectionsTest, DalWavePlay, DalWaveRecord	examples-cpp
Windows-CSharp	C# GUI apps. Visual Studio 2015 solution provided.	DalLoopbackApp, DalWavePlay, DalWaveRecord	example-csharp

Mac Package

Example Set (reference name used in this document)	Description	Applications Included	Source Directory
Mac-CPP	C++ command line apps. Makefile, CMake file and XCode project provided.	DalLoopbackTest, DalConnectionsTest	examples
Mac-ObjC	Objective-C GUI apps. XCode project provided.	DalLoopback	example/DalUIExamples

Before attempting to build any of the example projects, you will need to copy your `access_token.c` into the required directory, as shown in the following table.

Example Set	Access token directory
Windows-CPP	examples-cpp
Windows-CSharp	examples-csharp
Mac-CPP	examples
Mac-ObjC	examples/DalUIExamples/DalObjc

The table below summarises how to build each example set. Some example sets have multiple build options to show examples of different build systems. Only one needs to be used.

Example Set	Build Procedure
Windows-CPP	<ul style="list-style-type: none"> Visual Studio solutions: <ul style="list-style-type: none"> Multi-threaded runtime: <code>examples-cpp/vs2015-static/Examples.sln</code> Multi-threaded dll runtime: <code>examples-cpp/vs2015-dll/Examples.sln</code>
Windows-CSharp	<ul style="list-style-type: none"> Visual Studio solution: <ul style="list-style-type: none"> <code>examples-csharp/DalExampleApps.sln</code>
Mac-CPP	<ul style="list-style-type: none"> XCode project: <ul style="list-style-type: none"> <code>examples/xcode/DalExamples/DalExamples.xcodeproj</code> make: <ul style="list-style-type: none"> <code>cd examples && make</code> cmake: <ul style="list-style-type: none"> <code>mkdir -p examples/build && cd examples/build && cmake -G Xcode .. && cmake -build .</code>
Mac-ObjC	<ul style="list-style-type: none"> XCode project: <ul style="list-style-type: none"> <code>examples/DalUIExamples/DalUIExamples.xcodeproj</code>

C++ Examples

C++ examples are provided for both Windows and macOS.

The following table lists the arguments common to all the C++ example applications. Some of the applications have additional application-specific arguments that are provided in the relevant sections for each example.

Argument	Description
<code>-i=<INTERFACE></code>	Name or index of the wired Ethernet interface to use. If no interface is specified, the program will prompt the user to select one before continuing.
<code>-t=ptp</code>	Obtain the network time from the Precision Time Protocol (PTP) process.
<code>-t=rx</code>	Obtain the network time from the incoming audio packets
<code>-p=<PATH></code>	Path to the DAL binaries. Not required, defaults to current working directory
<code>-tx=<N></code>	Number of transmit channels to create for the application's Dante device. Default is 2.
<code>-rx=<N></code>	Number of receive channels to create for the application's Dante device. Default is 2.
<code>-n=<NAME></code>	Name of the application's Dante device. Default is application specific.

DalLoopbackTest

The DAL Loopback test application is a simple application for creating a Dante audio device. By default, the audio device is configured with 2 receive (Rx) and 2 transmit (Tx) channels.

The example will loop back all audio received on the Rx channels and transmit it out via the Tx channels. This device can be used to test basic DAL audio functionality on a Dante network.

The DalLoopbackTest example includes a complete implementation of a software-based Dante audio device. The port, model, manufacture information is provided as an example. It is expected that your application will use different configuration information.

The DalLoopbackTest example program has all the common arguments, and no extra application-specific arguments.

DalWavePlay

The DAL Wave Play example demonstrates how to play a WAVE file out to a Dante network. Once started, the application will appear as a Dante device on the network, and automatically send the WAVE file audio to any devices subscribed to the application's transmit channels.

If the application is started with more transmit channels than the number of audio channels in the WAVE file, audio will only be present in the first N Dante transmit channels, where N is the number of WAVE file audio channels. Conversely, if there are more audio channels in the WAVE file than the DAL device transmit channels, then only the first N channels of the audio file will be transmitted, where N is the number of DAL device transmit channels.

The sample rate of the WAVE file must match the sample rate configured in DAL by the application. If there is a mismatch, the application will terminate at start-up.

The application currently supports WAVE files with 16, 24 and 32-bit PCM audio encodings. The file's audio encoding will be converted by the application to the native DAL shared memory audio encoding, which is 32-bit PCM. The application will terminate at startup if the WAVE file contains an unsupported audio encoding.

The application will terminate once the entire WAVE file has been played.

The DAL Wave Play example has the following extra application specific arguments.

Argument	Description
<code>-f=<FILENAME></code>	Name of the WAVE file to play

DalWaveRecord

The DAL Wave Record example demonstrates how to record audio from the Dante network to a WAVE file. Once started, the application will appear as a Dante device on the network, and automatically start writing the network audio data to file. The WAVE file will be completed once the application is terminated.

The resulting WAVE file will have:

- Number of audio channels = DAL configured number of receive channels
- Sample rate = DAL configure sample rate
- Encoding = PCM with bits-per-sample determined by command line arg (`-s`). The application will convert from the DAL shared memory PCM32 encoding to the selected WAVE file encoding.

The DAL Wave Record example has the following extra application specific arguments.

Argument	Description
<code>-f=<FILENAME></code>	Name of the WAVE file to record to in the current directory
<code>-s=<BPS></code>	Bits per sample to use for the WAVE file audio data. The default is 16.

DalConnectionsTest

The DAL Connections Test example demonstrates the features of the Connections API. The program will attempt to connect to an already running DAL instance. This could be either the loopback test or another DAL-enabled application.

The program adds callbacks to the Connections API, and displays changes to the available sources, available destinations and local channel connection state.

The DAL Connections Test example program is run with the following arguments:


Argument	Description
<code>-a=<port num></code>	The ArcpLocal port number in use by the DAL instance
<code>-c=<port num></code>	The ConMon client port number in use by the DAL instance
<code>-d=<port num>/<path></code>	The Domain Client port number or path in use by the DAL instance
<code>-m=<port num></code>	The MDNS mDNS Client port number in use by the DAL instance Windows Only

Console commands allow the exercise of various Connections API commands and queries:

Command	Description
<code>help</code>	Print usage
<code>lists</code>	List the available sources for a given receive channel
<code>listd</code>	List the available destinations for a given receive channel
<code>sets</code>	Set or clear the remote transmit channel from which a local DAL receive channel should receive audio. The Connections API will create, update or remove a Dante subscription on the local receive channel
<code>setd</code>	Set or clear the remote receive channel that should receive audio from the local DAL transmit channel. The Connections API will create, update or remove a Dante subscription on the remote receive channel
<code>rxstate</code>	Show the current connection state for a given Rx channel
<code>txstate</code>	Show the current connection state for a given Tx channel
<code>id</code>	Identify a device, for example by flashing an LED. This is only supported on some Dante devices and the implementation is device dependent; listd or lists indicate devices that support identification.

Command	Description
<code>clear</code>	Clears all remote Rx channels subscriptions on the local device that are not part of a currently configured destination. It will not remove any subscriptions that are required for the current set of connections.


Windows C# Examples


 **Important:** C# example code is provided for rapid prototyping purposes - however, for managed code, garbage collection will cause a glitch if it occurs for more than 1 to 2 milliseconds at a time. Accordingly, the C# example code is not likely to be suitable for production software. If using C# for your software, Audinate recommends keeping the Dante audio runtime code in a separate process, using a C++ back end.

Dal Managed Library

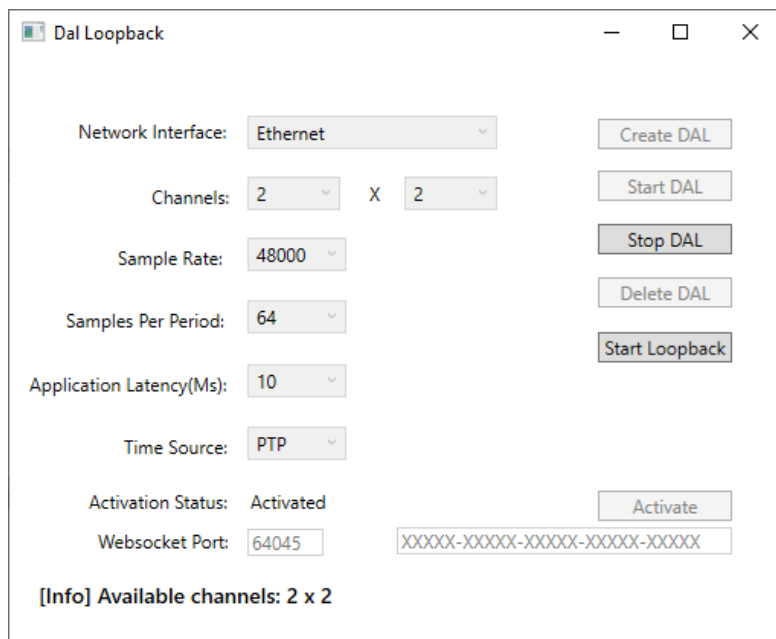
All C# examples share a common C# managed library. To allow C# access to the C++ examples, we have created a Common Language Runtime (CLR) managed DLL that allows easy access for the following C# examples.

While a common DLL has been provided in the DAL examples for ease of sharing between the different programs, it is not recommended to make a common managed DLL that can easily access the DAL C++ routines. We recommend keeping your software secure by limiting the re-usability of any managed C++ code you create. For example, this could expose your private access token to potential unauthorised reuse by third parties.

 **Note:** Not all functionality available in the DAL C++ API has been replicated in the C# managed API - only the APIs necessary for the following examples.

 **Note:** The C# example code is provided for rapid prototyping purposes - however, for managed code, garbage collecting can cause a glitch if it occurs for more than 1 to 2 milliseconds at a time. As such, the C# code example is not likely to be suitable for production software. If using C# for your software, we recommend keeping the Dante audio runtime code in a separate process using a C++ back end. We also recommend using the latest .NET libraries to help with better garbage collection.

Dal Loopback



The DAL Loopback C# example is conceptually similar to the C++ Loopback example. It is a simple application for creating a Dante audio device that will loop back all audio received on the Rx channels, and transmit it out the Tx channels. This device can be used to test basic DAL audio functionality on a Dante network.

The WPF (Windows Presentation Foundation) based GUI allows the user to select a valid network interface, the number of Rx and Tx channels, samples per period, application latency (NB: not Dante network latency) and the time source. Additionally you can set the WebSocket port to a known port number, or leave as 0 to use an ephemeral port.

After running the C# example, the DAL device will be initialized, but not started. In this state it will not appear in Dante Controller (DC). When the **Start DAL** button is clicked, it will appear in DC. Initially it will appear as unlicensed or not activated in DC if you are using an activation token. If you are using an evaluation token it will appear in DC, and the C# GUI in an activated state.

The audio loopback functionality will begin after you click the **Start Loopback** button. All audio received by DAL over the Dante network will now be looped back to the Dante network.

You can stop and start DAL multiple times - however, if you **Delete DAL** you must **Create DAL** again before you can start it. The **Delete DAL** button is included to show you how to clean up DAL before shutdown.

The running DAL status will appear as messages in the *statusTextBlock* at the bottom of the window.

After copying the `dante_activator_redist.exe` to the runtime folder (e.g. `examples-csharp\Loopback\bin\x86\Debug`) you will be able to activate the DAL device using an entitlement ID, via the Loopback GUI. Additionally, you can activate the DAL device by running the activator tool via your terminal. For running the Dante Activator tool from the command line, please refer to the [Command Line Dante Activator User Guide](#).

The **Activate** button is included to demonstrate how to activate a DAL device using the Dante Activator command line tool. You will need a valid entitlement ID to activate DAL. It will run the `dante_activator_redist` executable with the following command:

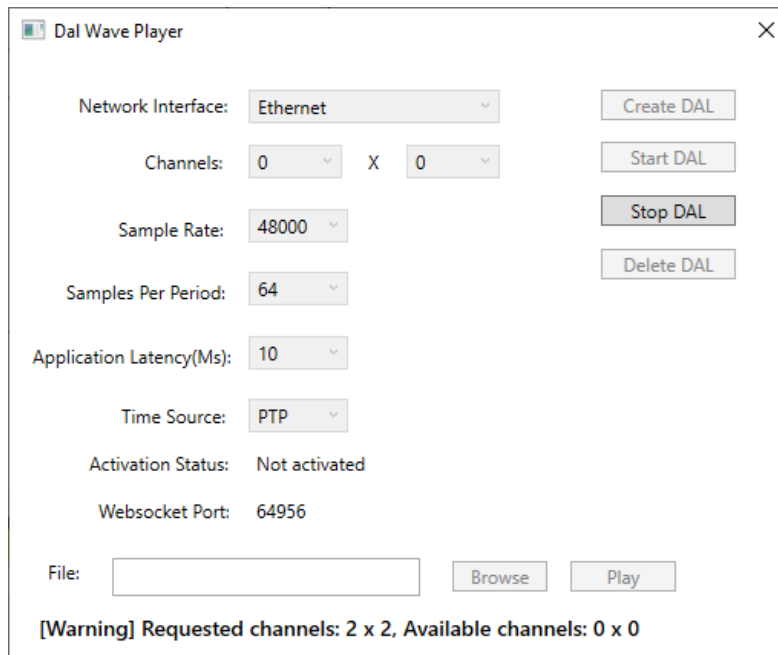

```
dante_activator_redist.exe activate --w=<websocket number> --  
e=<entitlement ID>
```

- where <websocket number> and <entitlement ID> are both obtained via the GUI.

Activating a DAL device can take a few seconds while it contacts the Audinate software license server. If activation fails for any reason, a message window will appear showing an appropriate error message.

For more information on Activation and DAL please refer to [Activation](#).

Dal Wave Player



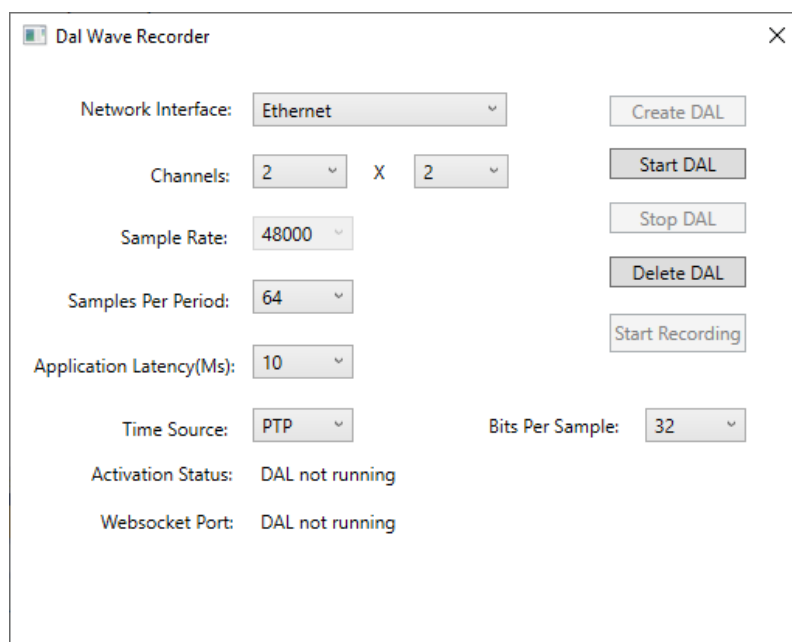
DAL Wave Player has a similar graphical interface to the DAL Loopback application. Again, you can start and stop DAL; delete and create DAL; and select a valid interface, the number of Rx and Tx channels, samples per period, application latency (NB: not Dante network latency) and time source. Status information on the application will appear in a text field at the bottom of the window.

The WebSocket port field cannot be set by the user, and will therefore start DAL with the WebSocket set to open on an ephemeral port. There is no **Activate** button or field for the activation ID. This has been left as an exercise for the reader.

To play a wave file to the Dante network, DAL must first be started and activated. Click **Browse** and select a compatible wave file. The C# example only supports wave files with up to 2 channels of audio.

The wave file will loop from the beginning until you stop playing, or shut down the DAL application.

Dal Wave Recorder



Dal Wave Recorder

Network Interface: Ethernet

Channels: 2 X 2

Sample Rate: 48000

Samples Per Period: 64

Application Latency(Ms): 10

Time Source: PTP

Bits Per Sample: 32

Activation Status: DAL not running

Websocket Port: DAL not running

Create DAL

Start DAL

Stop DAL

Delete DAL

Start Recording

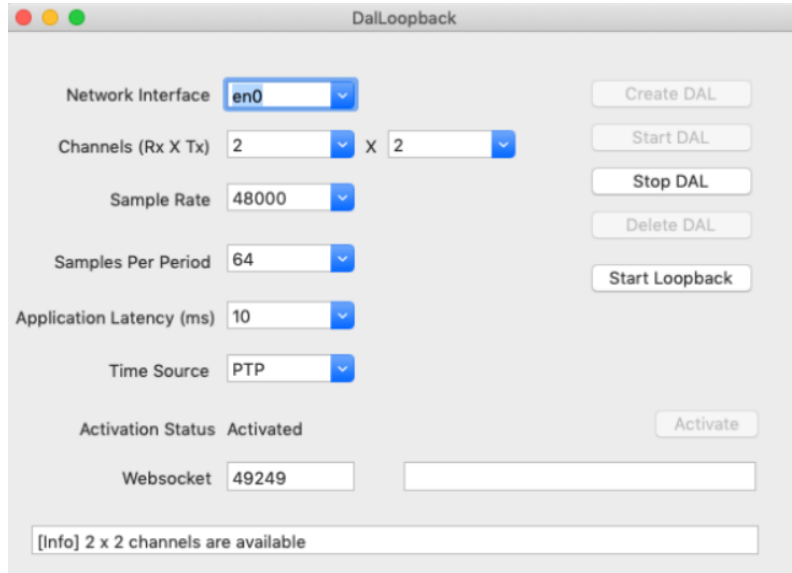
DAL Wave Recorder has a similar graphical interface to the DAL Loopback application. Again, you can start and stop DAL; delete and create DAL; and select a valid interface, the number of Rx and Tx channels, samples per period, application latency (NB: not Dante network latency) and time source. Status information on the application will appear in a text field at the bottom of the window.

To record a wave file to disk, you must first start DAL. After starting DAL you can click **Start Recording**. A standard Windows file 'Save As' dialog will appear. Select a file name and click **Save**. All audio received by the DAL device will now be saved to disk in the chosen wav file.

The file will continue recording until you click **Stop Recording** or shut down the application.

macOS Objective-C Examples

Dal Loopback App



The DAL Loopback Objective-C example is functionally equivalent to the corresponding C# example. Refer to [Dal Loopback](#) for a description of the common functionality.

To be able to activate the DAL device via the application, the `dante_activator_redist` binary needs to be copied to the runtime folder

`examples/DalUIExamples/DalLoopback/Debug/DalLoopback.app/Contents/MacOS.`

Again, refer to [Dal Loopback](#) for details on how the Dante device can be activated, either via the Loopback GUI, or separately on the command line.

Adding DAL to Your Application

The DAL package includes header files and libraries for DAL and its supporting packages.

Preparation

- Choose a distinct set of port numbers or Unix paths that DAL can use for its protocols within your application. DO NOT use the numbers specified in the example code
- Choose a host directory for the DAL and Dante Activator binaries
- Choose a writable host directory for DAL configuration files
- Choose a writable host directory for log files
- Choose a writable host directory for activation files

All the above will need to be passed to DAL via the DAL instance configuration.

Compiling

- Add the DAL package "include" directory to your project include paths
- Include one or more of the following component headers as needed:
 - `"audinate/Dal/Instance.hpp"`
 - `"audinate/Dal/Audio.hpp"`
 - `"audinate/Dal/Connections.hpp"`
- Add the supplied `access_token.c` to your application source tree

Linking

The DAL package includes all required libraries for an application to include DAL. On Windows, libraries are provided for 32-bit and 64-bit platforms, and for Debug and Release configurations. On macOS, only the Release configuration is supplied.

The following libraries are required for any part of the DAL API.

- `Dal.lib/.a`
- `dal_api.lib/.a`
- `libprotobuf-lite.lib/.a` (Debug builds) or `libprotobuf-lite.lib/.a` (Release builds)
- `sodium.lib/.a`

Application Packaging and Installation

In addition to the application's own packaging and installation requirements, the following DAL-specific files need to be packaged and installed:

- The DAL shared libraries (contained in the DanteApplicationLibrary package)
 - Windows: `dal.dll`
 - Mac: `libdal.dylib`
- The DAL support binaries (contained in the DanteApplicationLibrary package)
 - Windows: `apec3.exe`, `common_server.exe`, `mDNSResponder.exe` and `ptp.exe`
 - Mac: `apec`, `common_server`, `ptp`
- The Dante Activator binary, if DAL activation is required (contained in the Dante-Activator package).
 - Windows: `dante_activator_redist.exe`
 - Mac: `dante_activator_redist`

Running

DAL instances rely on a shared library and several child processes to implement different aspects of Dante device functionality. The library will execute the child processes as needed. The shared library should be placed in the same directory as the host application binary. The DAL Instance must be configured with the path to where the child processes can be found. If not otherwise specified, the child processes are assumed to be in the same directory as the host application.

Network Interface Selection

Before starting a DAL instance you must select a valid Ethernet interface. Dante audio devices are designed to run on wired networks. A valid Ethernet interface therefore does not include Wi-Fi or virtual interfaces. The DAL API call `getAvailableInterfaces()` is provided for you to discover the interfaces that DAL can use.

A Network Interface can be identified by either a string name or an index. It is recommended, and demonstrated in the examples, to display the string name of the network interface to your users. The index is defined by the operating system and is not typically identifiable by your customers.

Activation

When using the activation enabled token, each DAL software instance must be activated on the host system before DAL becomes fully operational. When in the unactivated state, a DAL activation will appear in Dante Controller as an unlicensed device with red text. Activation only needs to be performed once for each software install.

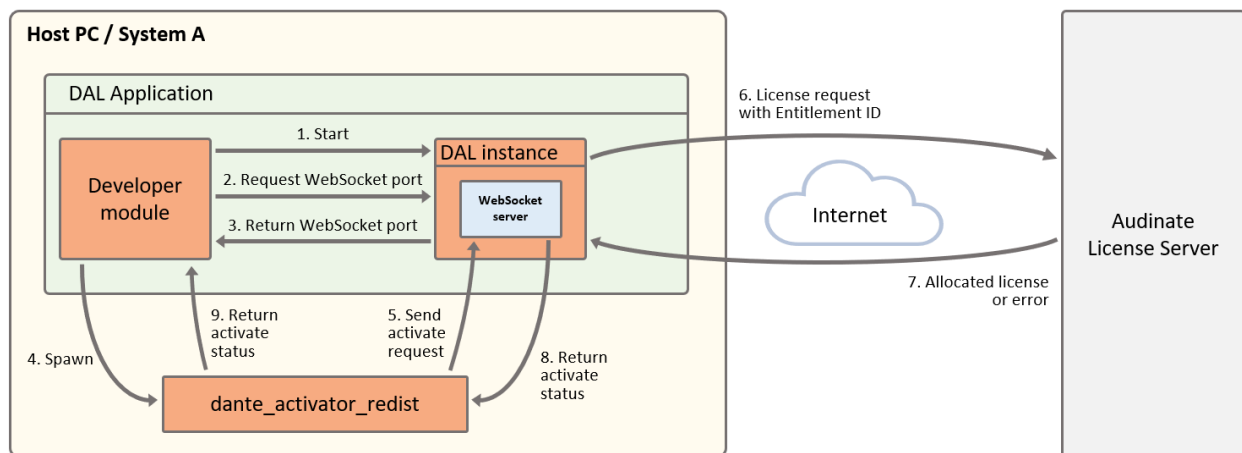


Figure 2 - DAL Activation Block Diagram

The diagram in **Figure 2** shows a high-level view of the main modules and operational flow involved in DAL activation. Information at the more detailed programmatic level, including DAL activation APIs, are described in [Activation](#).

The blocks shown in the diagram are:

- Host PC / System: The physical machine that the DAL application runs on.
- DAL Application: The entire software application. Includes the modules written by a Developer and all components provided by DAL.
- Developer Module: A module written by a Developer that controls DAL and any other functionality specific to the application.
- dante_activator_redist: The Dante Activator command line executable.
- Audinate License Server: Audinate server reachable via the Internet that DAL devices communicate with to perform activations.

As a DAL software developer the most important aspects are the flows shown from and to the Developer Module. The other flows are implemented by Dante software, and don't require direct Developer involvement, but are included to illustrate the complete end-to-end DAL activation system.

- 1: The Developer Module creates DAL and starts a DAL instance. Internally the DAL instance starts a WebSocket (WS) server.
- 2 - 3: The Developer module requests and receives the WebSocket port number from the DAL instance. This is needed to be passed to dante_activator_redist.
- 4: The Developer module spawns the dante_activator_redist executable as a separate process. The WebSocket port number and an entitlement ID are passed as command line arguments. See [Activation](#) for more details on the Dante Activator command line tool usage and arguments.
- 5: dante_activator_redist sends a WS activate request to the DAL WS server.
- 6-7: The DAL instance communicates with the Audinate License Server to request and receive an activation (or error).
- 8-9: The activation result is propagated back from DAL to dante_activator_redist to the Developer module.

It is expected that all files in the DAL persistent directory will be retained after a system upgrade or reset.

Using DAL

Instance Management

Applications can configure many aspects of the DAL instance. These parameters are configured using the `InstanceConfig` class. This object is then used to create an Instance object. The configuration is fixed for the lifespan of the Instance object.

Once the instance is running, its configuration can be managed programmatically or via network control applications such as Dante Controller.

Child Processes

DAL instances rely on several child processes. Of these, some processes are always needed, while others are only required for certain modes of operation.

Child Process	Role	Required
common_server.exe	Provides Dante control & monitoring functionality	Always
apec3.exe	Responsible for audio routing management	Always
Ptp.exe	Track network time via the PTP protocol	When using PTP as the TimeSource
mDNSResponder.exe Windows Only	Provides a per-instance mDNS discovery process	When the Audinate mDNSResponder service is not running. Windows Only

The DAL instance needs to know the path to the supporting binaries. The `InstanceConfig` class includes a binary path configuration parameter for this purpose.

Once running, the DAL instance is responsible for starting and monitoring the child processes. By monitoring the child processes, DAL can restart any processes that terminate unexpectedly. Conversely, if a child process loses contact with the host application, they will terminate themselves. The Instance API includes a callback to inform the host application of the current state of each child process. Possible status values are as follows:

Status	Description
Stopped	The instance is stopped. No child processes are running.
Unused	The instance is running but the given child process is not required
Pending	The instance is running. The child process is required but is not currently running.
Running	The instance is running, and the child process is running.

Activation

The following table lists the main DAL APIs relevant for activation. For brevity it is assumed that the Audinate::DAL namespace has been brought into scope.

API	Description
<code>InstanceConfig::setActivationDirectory</code> <code>InstanceConfig::getActivationDirectory</code>	Set and get the directory where the activation files will be stored. The activation files are written at activation time, so the directory needs to be writable.
<code>InstanceConfig::setProtocolSocketDescriptor</code> with <code>Protocol::WebSocket</code>	Set the WebSocket server network port number. This can be set to 0 to use an ephemeral port, or non-zero for a fixed port.
<code>Instance::getProtocolSocketDescriptor</code> with <code>Protocol::WebSocket</code>	Get the WebSocket server port number. The WS port number is needed during activation, as it will be passed to the Dante Activator tool on the command line. This call will only return a valid value after the DAL instance has been started, as that is when the WebSocket server is started. It will return either the actual ephemeral port that has been selected, or the fixed port number that was requested in <code>setProtocolSocketDescriptor</code> .
<code>Instance::isDeviceActivated</code>	Check whether DAL is activated. This can be called both before and after the DAL instance has started. It can be used to change the behaviour of the application based on the activation status (e.g. prompt for activation).
<code>InstanceEvent::DeviceActivationStatusChanged</code>	An event of this type will be generated if the DAL activation state changes. In practice that will be when DAL goes from the unactivated to activated state.

There is currently no direct DAL API to perform the activation. The application code needs to spawn the Dante Activator command line tool (`dante_activator_redist.exe` for Windows and `dante_activator_redist` for Mac) and pass it the DAL WebSocket port and the entitlement ID to be used for the activation. Dante Activator is a general tool applicable to DAL as well as other use cases. A full description of its usage can be found in the separate [Command Line Dante Activator manual](#). For the purposes of DAL, only the following invocation is needed to perform DAL activation:


```
dante_activator_redist -w=<ws_port_num> --e=<entitlement_id> activate
```

As described in [Application Packaging and Installation](#), the `dante_activator_redist` binary needs to be installed into the same location as the other DAL binaries on the host machine.

The general code flow that the application needs to implement for activation is as follows:

1. Call `InstanceConfig::setActivationDirectory` to point to a writeable directory.
2. Optionally call `InstanceConfig::setProtocolSocketDescriptor(Protocol::WebSocket, SocketDescriptor(ws_port_num))` to set the WebSocket port number. It is recommended to set the port number to 0 to request use of an ephemeral port. If this is not explicitly set, the default is 0.
3. Optionally call `Instance::isDeviceActivated` to determine whether DAL has been activated so as to determine whether to proceed to activation logic or normal application start-up.
4. Call `Instance::start` to start DAL.
5. Call `Instance::getProtocolSocketDescriptor(Protocol::WebSocket)` to get the port number actually used by the WebSocket server after it has started.
6. Spawn `dante_activator_redist` with the WebSocket port number and entitlement ID arguments as shown above to perform the activation.
7. Get the status of the activation from the return value of the spawned process. 0 is success, and any other value is failure. On failure there may also be an `activationFailed.txt` file written in the same directory as the binary.
8. Optionally wait for the `InstanceEvent::DeviceActivationStatusChanged` to be notified that the activation has successfully completed.

Audio Transfer

The DAL Audio API provides access to the instance's internal audio buffers. Each transmit and receive audio channel has its own buffer in memory, consisting of several samples divided into chunks or periods.

DAL tracks network time via PTP or received audio packets. As time passes, blocks of audio become available for reading or writing. When this happens, the audio API fires a callback to notify the application that the samples are available.

Applications that process audio synchronously in the callback must read the next block(s) of received samples and write the next block(s) of samples to be transmitted before returning from the callback. Applications that process audio asynchronously must have transmit samples ready before returning from the callback, and can read received samples after the callback has completed.

Each channel buffer is a circular buffer. Once the buffer has wrapped back to the same location, old samples are lost. Applications control both the number of samples available in each block (samples per period) and the number of blocks in each buffer (periods per buffer). Increasing the number of blocks in the buffer increases the time before samples are overwritten - at the cost of using additional memory.

Connection Management

The Connections API provides an optional simple mechanism for an application to make audio connections between the local DAL instance and other devices in the network.

Note that other mechanisms also exist for Dante connections management, control and monitoring purposes. End users commonly manage Dante connections via Dante Controller, and some Dante

equipment manufacturers or software developers have implemented their own controllers for specific use cases. It is important to consider your application's use of the Connections API in the context of wider Dante networks of other devices, so that end users don't experience unexpected behaviours due to the presence of multiple controllers on the network.

The Connections API may be subject to further enhancement in future DAL updates in response to ISV or end-user needs. A summary of the current API is listed below. More details and the most up-to-date documentation for the API calls can be found in the SDK [Connections.hpp](#) file.

API Calls	Description
<code>getLocalDevice</code>	Get information about the local DAL device
<code>getAvailableSources</code> <code>getAvailableDestinations</code>	Get list of device channels that can be used as the source for a local receive channel, or destination for a local transmit channel
<code>setReceiveChannelSource</code> <code>setTransmitChannelDestination</code>	Set the source or destination channel for a local receive and transmit channel respectively
<code>getReceiveChannelState</code> <code>getTransmitChannelState</code>	Get the connection state of a local channel
<code>clearRemoteSubscriptionsToLocalDevice</code>	Clear all remote subscriptions to any of the local transmit device channels that are not made by the Connections API
<code>setLocalDeviceChangedFn</code> <code>setAvailableChannelsChangedFn</code> <code>setConnectionChangedFn</code>	Register callbacks for various events
<code>identify</code>	Sends a Dante "identify" message to a given device. Note that not all Dante devices support this message, and for those that support it, the action taken is device specific (e.g. blink an LED).

Connections API vs Embedded Dante API (eDAPI)

The Connections API functionality is deliberately limited. When more sophisticated control is needed, applications can use functionality from the Embedded Dante API (also known as Embedded DAPI or eDAPI), which is comprised of multiple APIs. These APIs provide greater control, but are substantially more complex than the Connections API. Contact [Audinate Sales](#) if you require more information.

Operational Overview

The Connections API models DAL as a set of local and remote receive and transmit channels.

For each local receive channel, the API maintains a list of all the available channels on the network from which audio could be received. The host application can create, update or clear that channel's source, and monitor that channel's connection status.

For each local transmit channel, the Connections API maintains a list of all the available remote channels on the network to which audio could be transmitted. The host application can create, update or clear that channel's destination, and monitor that channel's connection status. Note that a general Dante transmit channel can have many remote destinations, but the Connections API only models a single remote destination for each local transmit channel.

Dante networks are configured via subscriptions. A subscription specifies the name of the transmit channel from which audio should be received. The receiving device is responsible for negotiation with the transmitting device. Internally, the Connections API translates connections into subscriptions. For receive channels, the difference is minimal. For transmit channels, the Connections API must identify the remote device and maintain a connection to that device.

The Connections API monitors channels on which it has made subscriptions, both local and remote. If the API's subscriptions are changed (for example, using Dante Controller), it will attempt to re-create those subscriptions to match the connection configuration. When a connection is modified or removed, the API will attempt to clean up any subscriptions it has made during its current instantiation. Here "connection" refers to connections made by the Connections API itself, and not to any connections made by an external controller (e.g. Dante Controller). Specifically, any subscriptions to DAL's local transmit channels made by an external controller are not tracked by the Connections API, and are thus not restored nor cleared by the API.

The Instance API and Connections API can be used in the same or separate processes. When used together, they should share the same DAL object.



Note: The Connections API requires an ARCP port number to be specified. This number should match the "ArctpLocal" port number specified for the DAL Instance. If the instance's ARCP port number is used, the Connections API may not behave correctly when the DAL instance is enrolled in a Dante Domain.

Available Channels

The Connections API discovers channels on the network to which connections can be made. The list is filtered to exclude channels that are incompatible, such as channels with different sample rates. Channels are grouped by device. For each device, the Connections API provides several pieces of information:

- Name: The name of the device. This is needed to create connections.
- Address: The device's IP address on its primary Dante network.
- Identifiable: Whether or not the device supports the *identify()* command presented in the Connections API.
- Sending: Is the device currently sending any audio on the network (i.e. are other devices configured to receive from it and currently receiving its audio?)

Connections, Persistence and Cleaning up Subscriptions

The Connections API maintains internal state about the current known set of connections, while it is running - however, this state is not persistent across instantiations. When the API starts up, the internal state is empty which means it has no connections. It is important to understand that even if the DAL device itself has receive or transmit connections, the Connections API does not track those on start up. Every channel is considered as "unmanaged" at that point, until one of the Connections APIs (`setReceiveChannelSource` or `setTransmitChannelSource`) is invoked to set up a connection

for a channel. At that point it becomes “managed”, and the Connections API will actively ensure that connection is maintained.

When the API is stopped and restarted, the list of known connections is lost, but the underlying subscriptions may still be present on the network. The Connections API may be restarted, and different connections configured. For RX channels, the new connection subscription overrides the old one, as it replaces the subscription on the local RX channels.

For TX channel connections, the situation is more complex. The old connection’s subscription may still be present after the new one is configured. The new instantiation knows nothing about the old connection, and so does not clean it up before making the new connection. The channel may now be being sent to multiple destinations. If this process is repeated enough times, the local device may run out of transmit resources, leading to errors in the underlying subscription.

There are two ways to deal with this situation. The preferred solution is that the host persists the set of connections it has previously configured. When the Connections API is re-started, the persisted connections should be re-applied before any new connections are made or changed. The Connections API will clean up the old connections as part of applying the new ones.

There may be cases where this is not possible. For these situations, the API includes a function that will remove every subscription to the local device that it can find on the network that is not part of one of its current connections. This function will clean up all unwanted subscriptions previously made by the Connections API, but will also clean up any subscriptions made by other controllers, which a user may have wanted. As such, it should be used sparingly, or in cases where the Connections API is known to be the only controller on the network creating subscriptions to the DAL instance.

Note that this function is applied to all subscriptions that are known at the time the function is called. If the function is called immediately after starting the Connections API, there may be remote subscriptions that have not yet been discovered, because discovery is not instantaneous upon DAL start-up.

Events

The Connections API provides callbacks to notify the application when certain events occur. These events are:

Event Type	Description	Register Callback API
Local device changed	This event occurs when a property of the local device has changed.	<code>setLocalDeviceChangedFn</code>

Event Type	Description	Register Callback API
Available channels changed	<p>This event occurs when the Connections API discovers that a remote channel has been added to or removed from the list of available channels that the local DAL device can use as a source or destination. Example triggers include:</p> <ul style="list-style-type: none">▪ A remote device coming online, resulting in channels becoming available▪ A remote device going offline, resulting in channels becoming unavailable▪ A remote device changing its sample rate to one that is incompatible with the local device, thus removing its channels from the available list	<code>setAvailableChannelsChangedFn</code>
Connection state changed	<p>This event occurs when a connection state changes for one or more of the local channels. For example, this event may be triggered when a connection is made as a result of calling <code>setReceiveChannelSource</code> or <code>setTransmitChannelDestination</code>. In that case the event may fire multiple times and can be used to monitor the progress and result of the set channel operation. Another example trigger event is when a connection is broken for any reason, such as the remote device going offline.</p>	<code>setConnectionChangedFn</code>

Specific Behaviours

The Connections API is intentionally simple and limited, and as a result there may be aspects of the Connections API behaviour that may not be obvious to new users of the API, or experienced users coming from eDAPI.

Behaviour	Explanation
Connections API actively manages connections it has set	<p>Once a connection is made via the API, that connection becomes managed and its state is actively maintained. This means that any changes made by an external controller that modify that state will be restored by the API. Examples include:</p> <ul style="list-style-type: none"> External controller removes a receive or transmit connection made by the API. The API will automatically add the connection back. External controller replaces a receive or transmit connection made by the API. The API will remove the external controller's connection, and add back its own connection. External controller adds a connection to/from a channel that the Connections API has explicitly cleared. The API will remove the connection.
Connections API does not manage connections it has not set: pre-existing connections	<p>When the API is first started, any connections that already exist are not tracked by the API. Such pre-existing connections may have been set by an external controller, or by a previous run of the Controller API. Some specific implications include:</p> <ul style="list-style-type: none"> A pre-existing connection will not be reflected by any of the get APIs. Namely, <code>getReceiveChannelState</code> and <code>getTransmitChannelState</code> will show the channel connection status as 'None' even though the actual DAL device channel has a connection. Setting a transmit destination on a channel via the API will not affect any pre-existing transmit destinations on that channel. Specifically, the pre-existing transmit destinations will not be cleared. So that will result in multiple transmit destinations for the underlying channel, even though the API is only tracking the single one it has set. <p>Note: The <code>clearRemoteSubscriptionsToLocalDevice</code> API is provided to find and clear all unmanaged remote destinations. A similar clear API is not needed for the receive channels, as there is only one receive connection possible per channel, and those can be cleared by calling <code>setReceiveChannelSource</code> with empty string parameters.</p>

Behaviour	Explanation
Connections API does not manage connections it has not set: subsequent external controller connection changes	<p>This is similar to connections created before the Connections API is started, as described in the previous point, but has a few subtleties that should be highlighted - specifically, the different behaviours of receive and transmit channels bears further explanation.</p> <p>Each Dante receive channel only has at most one source, and hence only at most one connection associated with it. So once the Connections API has been used to set a receive channel connection, any external controller receive connection changes to that channel will be detected by the Connections API, and the API's own connection will be restored.</p> <p>Each Dante transmit channel can have many destinations. The Connections API only tracks at most one connection. So what happens when an external controller adds/removes/changes a connection on a Dante transmit channel depends on whether that is the connection being tracked by Connections API. As a reminder, the Connections API starts tracking a connection when it is used to set that connection (including to set it to empty - i.e. no connection). If an external controller makes any changes to the connection that the Connections API is tracking (i.e. same destination remote device and channel), the API will detect that and restore its own connection. If an external controller makes any other changes (e.g. adds another destination for the transmit channel) the Connections API will not track that, and thus will not interfere with that in any way.</p>
All channels are considered unmanaged until a connection has been set by a Connections API call	<p>This allows an application using the Connections API to pick just the channels it cares about, by explicitly setting them. All other channels are left untouched. Notably, if the Connections API is never called to set a connection on a receive channel, any connections made on that channel by an external controller will never be cleared or modified in any way by the Connections API.</p>

Server Socket Management

Dante instances use multiple protocols for both internal and networked communication. In many cases, the DAL instance acts as a server. These server sockets can't use well-known ports or Unix paths, as this would violate DAL's co-existence requirements. Instead, applications are responsible for selecting ports or Unix paths as appropriate for the DAL instance to use for each protocol.

Socket Descriptor types for each protocol are as follows:

Protocol	Windows	macOS
Domain Proxy Server / Client	UDP Port	Unix path
WebSocket Server	TCP Port	TCP Port
All other protocols	UDP Port	UDP Port

On Windows, all DAL sockets use UDP or TCP ports. On macOS, all externally-visible protocols and most internal protocols use UDP ports. The Domain proxy client port expects a Unix path.

Note: The example code in the DAL package demonstrates port configuration. However, applications must not copy the port numbers or paths used in the example programs, and should select their own ports to avoid clashing with other Dante applications running on the same computer.

Firewall Management (Windows)

The host application is responsible for firewall management, including the ports used by the DAL Instance.

The Windows firewall can be configured by application, or by port. When configuring by application, DAL child processes must also be allowed access. When configuring by port, the following ports must be opened:

Table 1 - DAL External Ports

Protocol	UDP Port Number	Required
PTP	319, 320	Required when using PTP as the time source
ARCP	Application Specified	Always
DBCP	Application Specified	Always
CMCP	Application Specified	Always
mDNSResponder	5353	Required when running a per-instance mDNS discovery process. Windows Only
ConMon Channels (Unicast)	Application Specified. Range of 9 ports starting from the specified base port	Always
ConMon Channels (Multicast)	8702, 8708	Always
Audio Ports	Application Specified. Range of channels starting from the specified base port	Always
Websocket	Application Specified / Ephemeral	Always

Command Line Activator

When using the command line activator for activating DAL, it must have access to the Internet - it will connect over secure HTTP access on port 443. The following URLs may be contacted by the tool:

- <https://software-license-oem-activator.audinate.com/>
- <https://infield-licensing-app.audinate.com>

Appendices

Building and Running the DAL Examples in Xcode

On macOS, the GUI example project files are generated using a CMake file. This section has been written to help those not familiar with CMake.

Requirements

- Xcode
- DanteApplicationLibrary binaries
- `access_token.c`
- CMake 3.17 or later to generate your own Xcode project files.

This process is tested in the following environments:

- Cmake 3.18 with Xcode 10.2.1 on macOS 10.15.7
- Cmake 3.20 with Xcode 12.0.1 on macOS 11.0.1

Instructions

Ensure you have the DanteApplicationLibrary binaries and your `access_token.c` file before building and running the examples.



Note: Replace version numbers where appropriate.

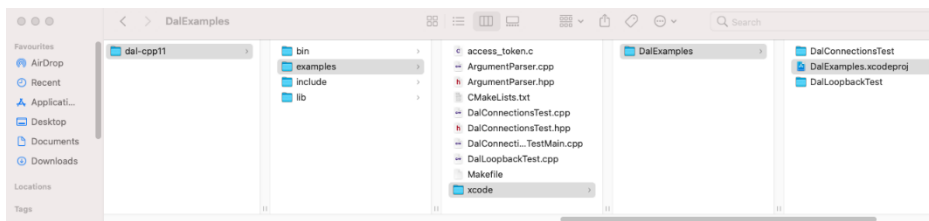
1. Decompress the DanteApplicationLibrary file and ensure that a `dal-cpp11` directory is created. This folder contains examples, and all other files required to build and run the examples. In this example we use `DanteApplicationLibrary-cpp11-[x.x.x.x]_macos.tgz`
2.

```
tar xvf DanteApplicationLibrary-cpp11-[x.x.x.x]_macos.tgz
cd dal-cpp11
```
3. Put your `access_token.c` file into the examples directory.
4. A default Xcode project file has been provided in the package - however, for better compatibility you can also generate a new project file for your Xcode version using CMake. To do so, create a build directory under the examples directory, change the current directory to build, then run CMake to generate Xcode project files.
5.

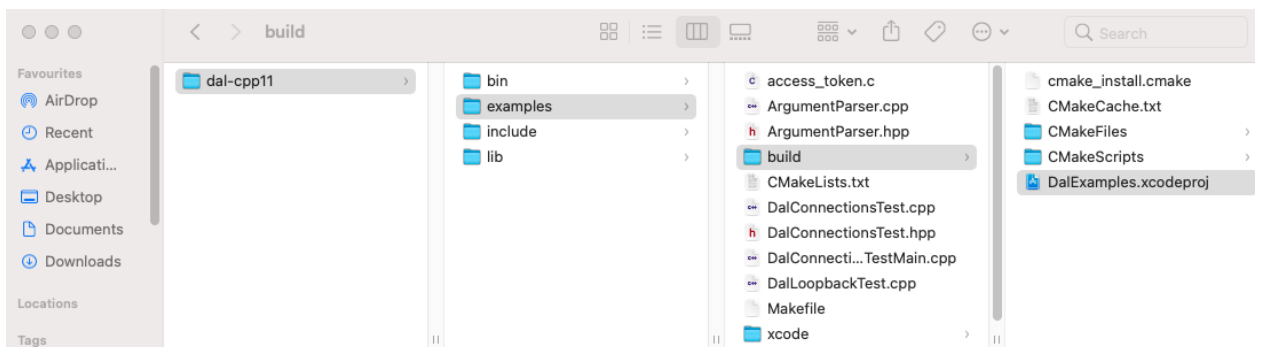
```
cd examples
```
6.

```
mkdir build
```
7.

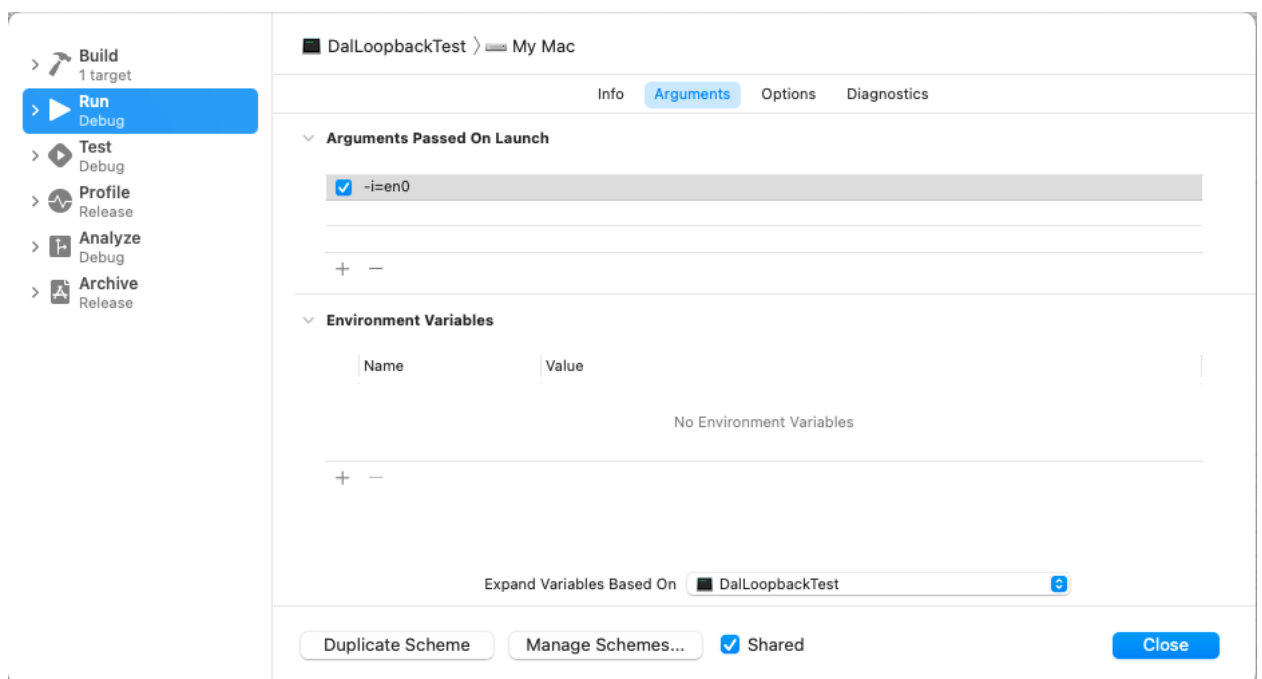
```
cd build
cmake -G Xcode ..
```
8. Open the Xcode project. If you are using the default Xcode project files in the package, select the following project.



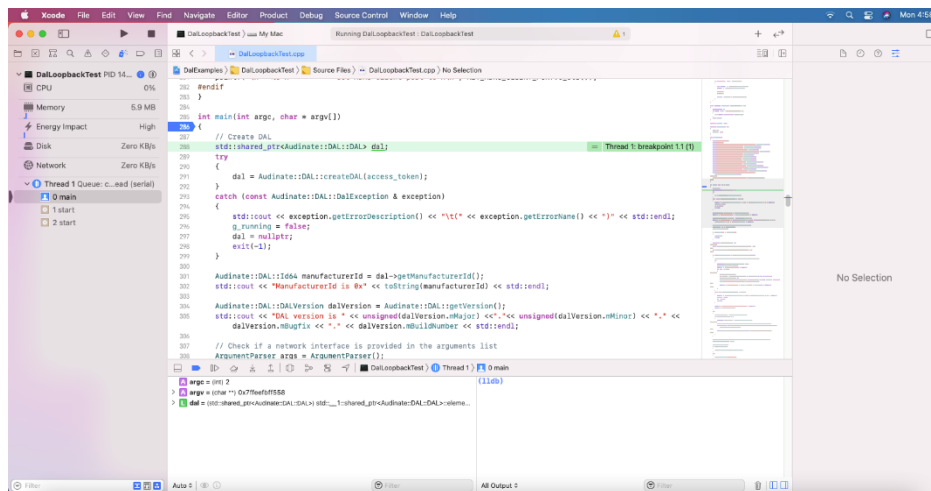
If you generated your own Xcode project files, select the following project.



- Set up the arguments you want to apply in the Scheme menu. Typically “-i=<interface>” is set to run more easily.



- You can now set breakpoints and/or continue to run in Xcode.



Notarize the DalLoopback Example Application on macOS

The following instructions have been provided as an example of how to notarize your macOS application. For general information on Notarization and macOS, please refer to https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution.

Prerequisites

- Audinate DAL package
- `access_token.c` - The unique access token Audinate generated for you
- Xcode and macOS command line utilities
- Keychain for signing
- Apple developer account for notarization

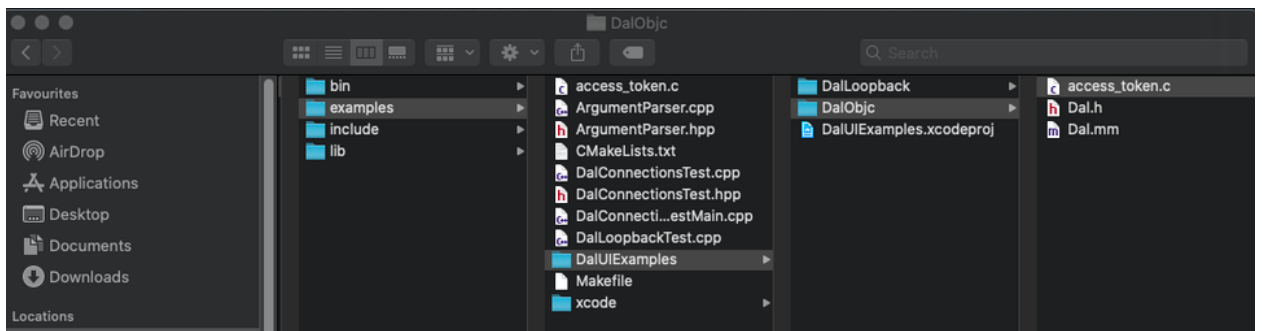
Audinate Signed Binaries in the DAL Package

- `libdal.dylib`
- `apex`
- `ptp`
- `common_server`

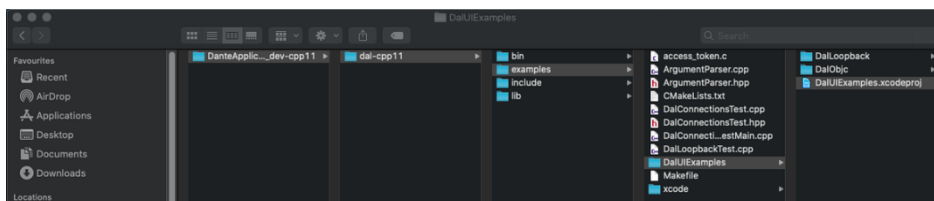
Instructions

Open the DalUIExamples project file

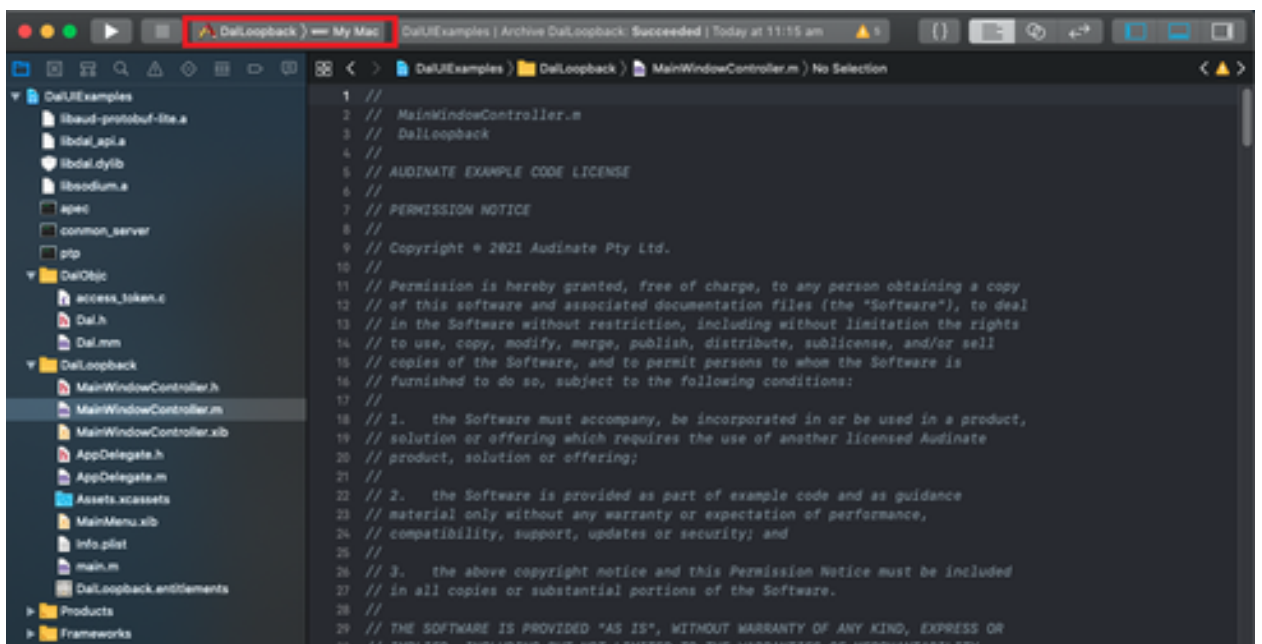
1. Ensure that your `access_token.c` file is under the `DalObjc` folder before opening the project file.



2. Click the DalUIExamples project file.

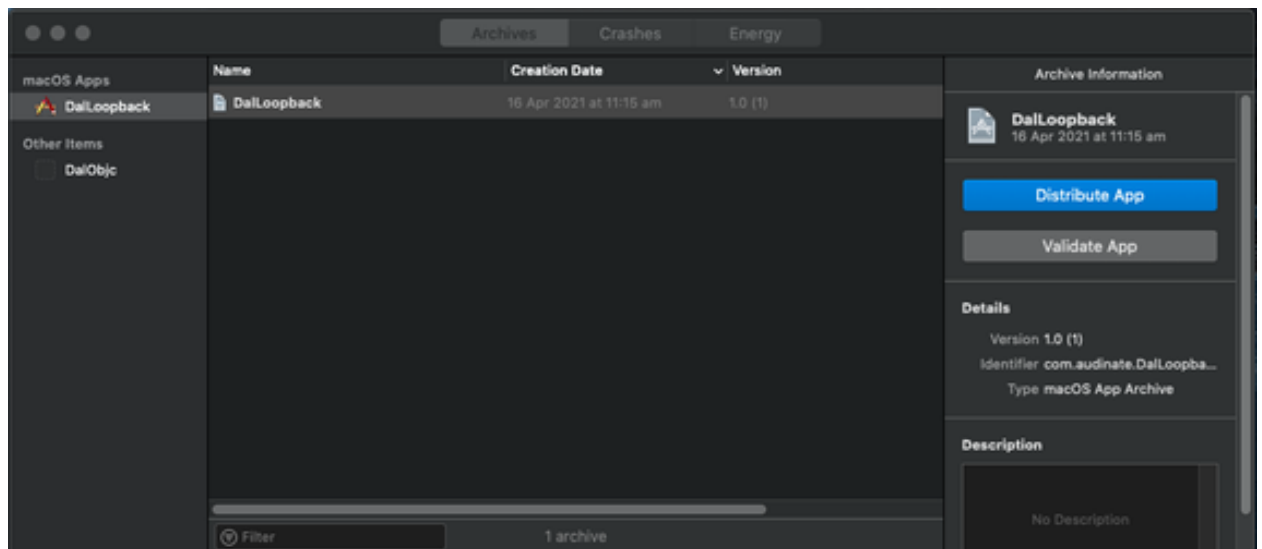


3. Ensure that the DalLoopback is selected as a target.

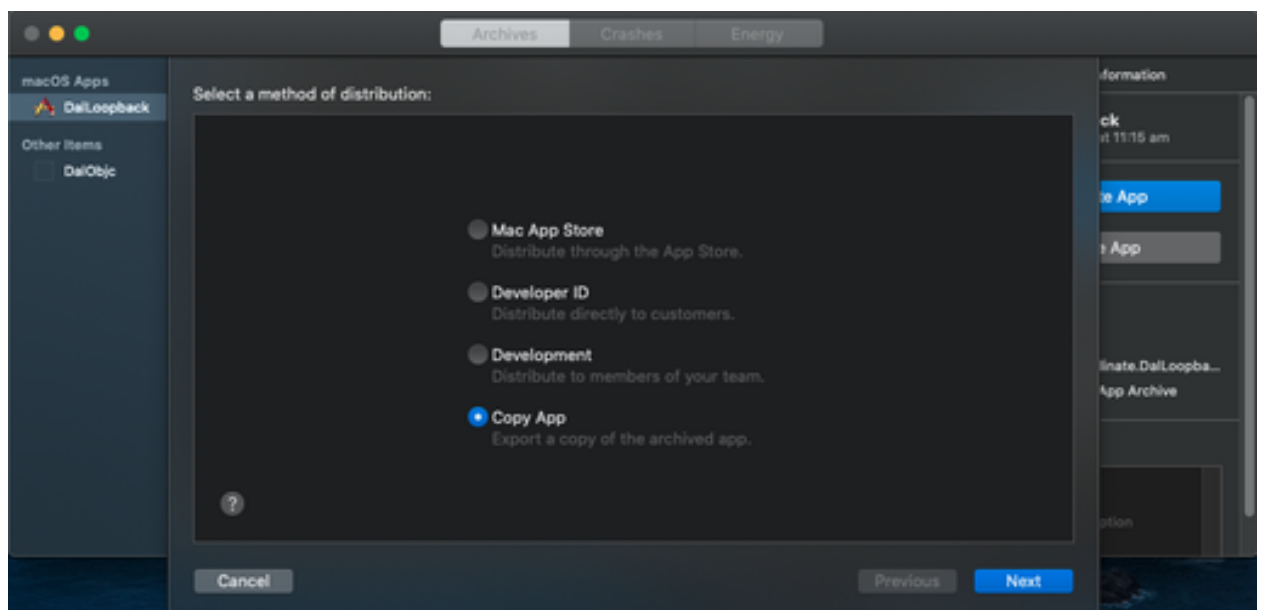


Archive the DalLoopback Application

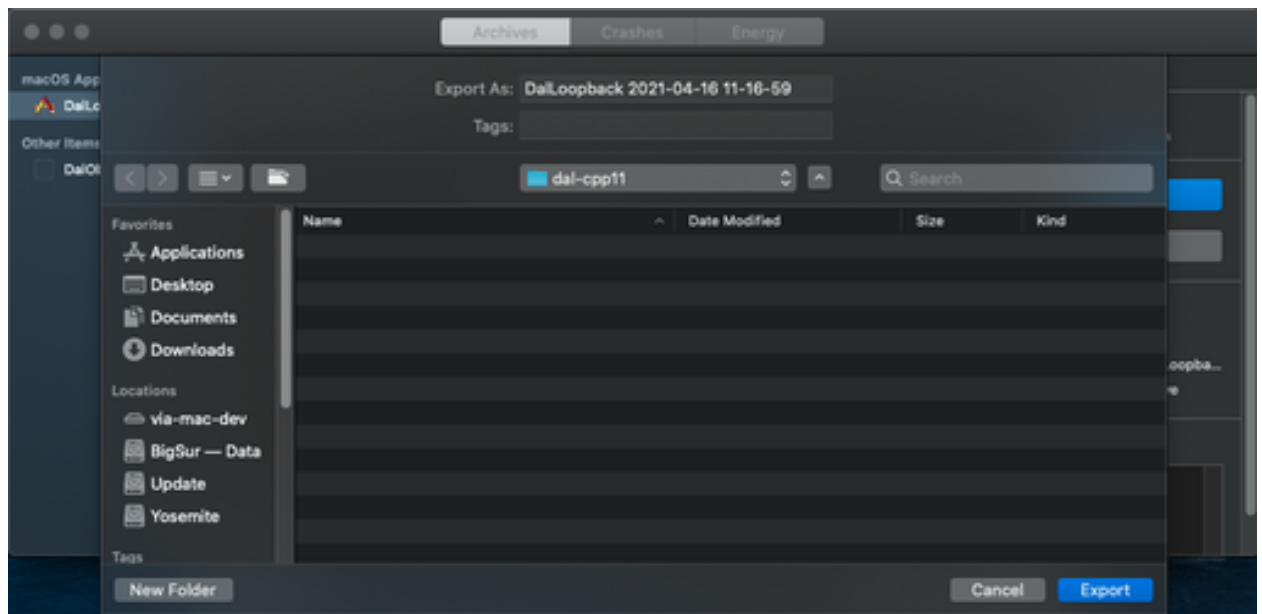
1. Run the archive by clicking the Product > Archive menu. You should see the following windows once the archive is done.



2. Click the Distribute App to copy the app into your workspace.



3. Select the location where you want to store the DalLoopback.app. This is the location where you will notarize the app.



Notarize the DalLoopback Application

The following step-by-step instructions show you one way to notarize the DalLoopback example application. You may have your own workflow for notarizing your existing applications.

1. Go to the folder where your archived copy of DalLoopback.app is.
2. Sign the DalLoopback.app with your keychain.

```
$ codesign --deep --force --verbose -o runtime --verify --
timestamp -s <identity> -v --keychain <keychain-filename>
DalLoopback.app
```

Where <identity> is the digital identity stored in your keychain and <keychain-filename> is the keychain filename for signing the app.

3. Ensure the DalLoopback.app is ready for notarization.

```
$ codesign -vvv --deep --strict DalLoopback.app
```

4. Create a dmg file for the DalLoopback.app.

```
$ hdiutil create -format UDZO -srcfolder ./DalLoopback.app
DalLoopback.0.0.1.99.dmg
```

5. Notarize the DalLoopback.app with your Apple developer account.

```
$ xcrun altool --notarize-app --primary-bundle-id <app-bundle-id>
--username <your-Apple-ID> --password <your-app-specific-password>
--file DalLoopback-0.0.1.99.dmg
```

Where <app-bundle-id> is the bundle ID for your DalLoopback app; “com.audinate.DalLoopback” is the default bundle ID in the DalUIExamples project; <your-Apple-ID> is your developer account, and <your-app-specific-password> is your password used to notarize the app.

You will get the RequestUUID back from the Apple server if the above command is successful. You can use the UUID to check the result of the notarization.

6. Ensure your notarization completed successfully. If not, visit the log file URL to see what’s wrong.

```
$ xcrun altool --notarization-info <RequestUUID> --username  
"<your-Apple-ID> --password <your-app-specific-password>
```

7. It is recommended to “staple” the notarization ticket to your .dmg file so that users don’t need to contact to the Apple server to verify it is Notarized.

```
$ xcrun stapler staple DalLoopback-0.0.1.99.dmg
```

8. Verify your final dmg file to ensure everything has completed successfully.

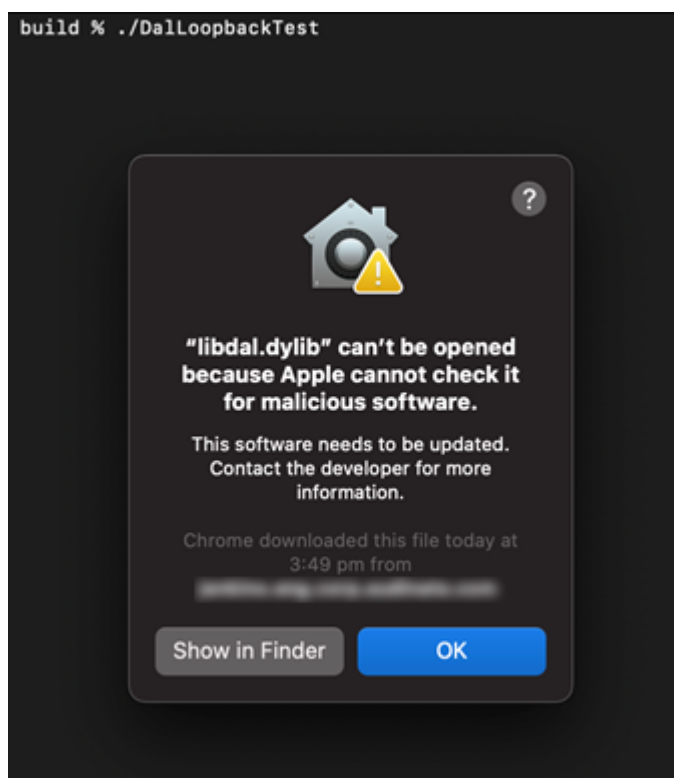
```
$ xcrun stapler validate DalLoopback-0.0.1.99.dmg
```

Load Disallowed DAL Library by System Policy on macOS

Library Disallowed Error

Apple continues to apply security restrictions to their operating system when creating your own software.

When trying to run the supplied DAL command line or UI examples or your own DAL-based command line applications for the first time on a macOS system, you might see the following error:



This error appears because macOS is not sure whether it is safe to run the DAL application package. For the command line application, a error like that shown below might also be presented:

```
macuser@MacBook-Pro build % ./DalLoopbackTest  
dyld[10770]: Library not loaded:  
@rpath/libdal.dylib
```

```
Referenced from:
/dal-cpp11/examples/build/DalLoopbackTest
Reason: tried:
'/dal-cpp11/examples/build/libdal.dylib' (code signature
in <44A8E071-0917-3976-B398-0B380C6FA261>
'/dal-cpp11/examples/build/libdal.dylib' not valid for use
in process: library load disallowed by system policy),
'/dal-cpp11/examples/build/libdal.dylib' (code signature
in <44A8E071-0917-3976-B398-0B380C6FA261>
'/dal-cpp11/examples/build/libdal.dylib' not valid for use
in process: library load disallowed by system policy),
'/usr/local/lib/libdal.dylib' (no such file), '/usr/lib/libdal.dylib'
(no such
file)
zsh: abort ./DalLoopbackTest
```

Reason for the Error

The DAL package might have been downloaded over the Internet from the Audinate portal using Safari or Chrome. The application or macOS might add a quarantine extended attribute to the DAL package, so that it can be blocked from running without user approval. You can check whether the quarantine attribute is set on the package by running the following command (adjusting version numbers as required):

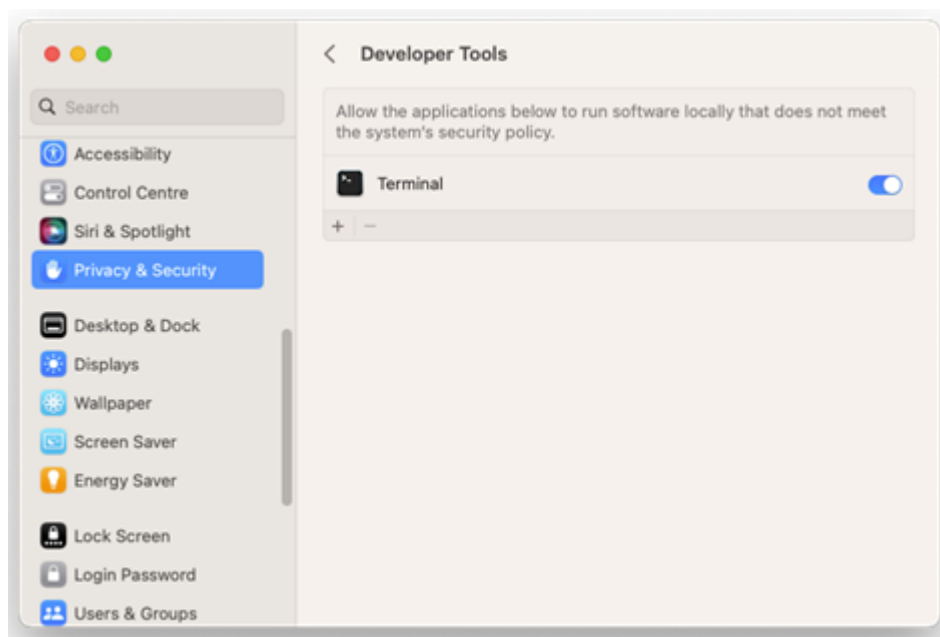
```
xattr DanteApplicationLibrary_cp-cpp11-1.2.0.0_macos.tgz

com.apple.macl
com.apple.metadata:kMDItemWhereFroms
com.apple.quarantine
```

To avoid the above issue when running DAL macOS applications such as DALoopbackTest, remove the attributes added by the application before decompressing the DAL package, as shown below:

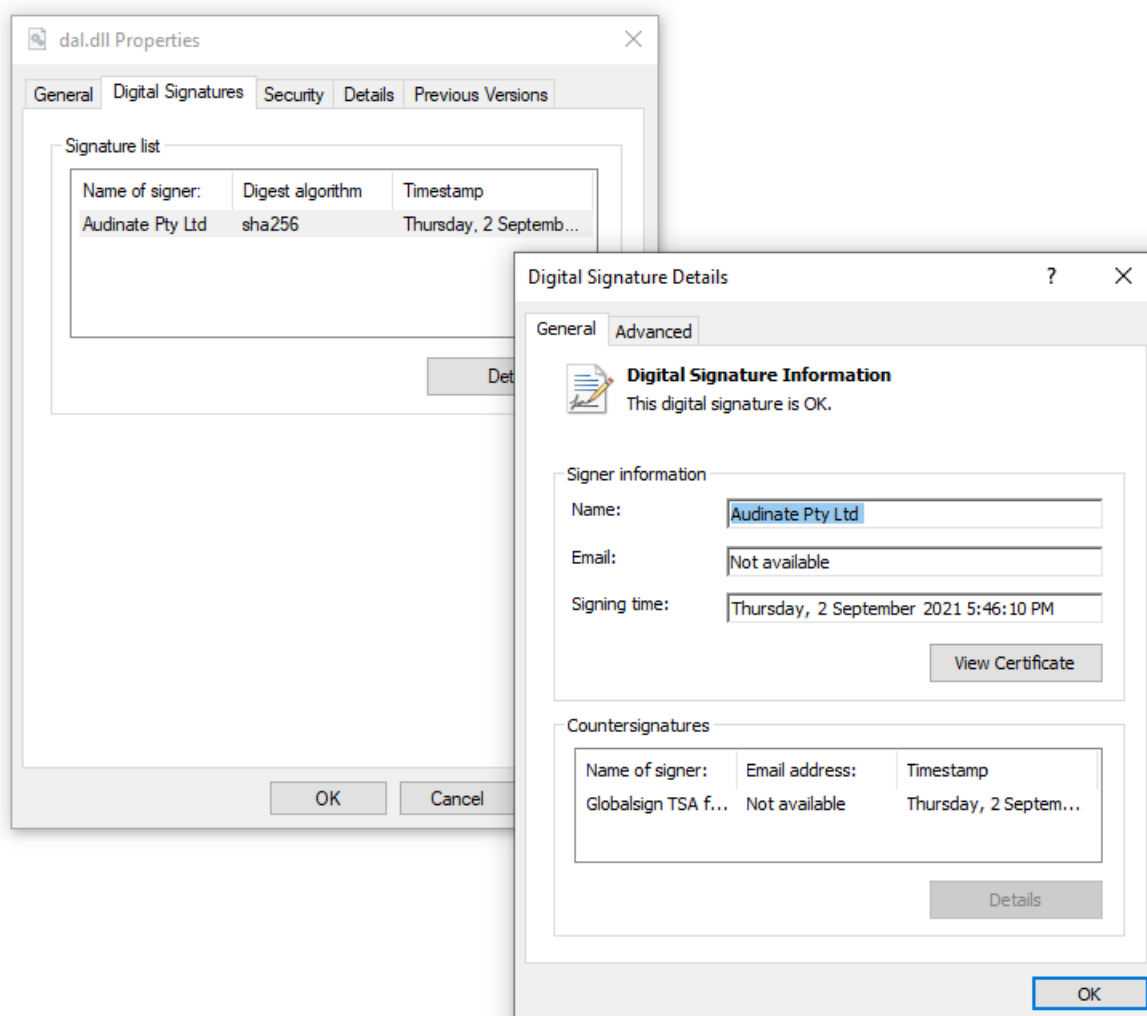
```
macuser@MacBook-Pro Downloads % xattr -c DanteApplicationLibrary_cp-
cpp11-1.2.0.0_macos.tgz
```

When running applications from the Terminal, another option is to turn off all security requirements. You can do that in System Settings > Privacy & Security > Developer Tools (allow Terminal to 'run software locally that does not meet the system's security policy').



Authenticating DAL Binaries on Windows

On Windows the DAL DLLs and executables are signed with an Audinate certificate. You can inspect the certificate by looking at the properties of the supplied binary files.



When using the provided DLLs and binary executables in your application you can confirm they are validly signed by Audinate and have not been tampered with.

This can be done using the Windows developer tool SignTool.exe:

<https://docs.microsoft.com/en-us/windows/win32/seccrypto/using-signtool-to-verify-a-file-signature>

or programmatically:

<https://docs.microsoft.com/en-us/windows/win32/seccrypto/example-c-program--verifying-the-signature-of-a-pe-file>

Glossary

A

ARCP

Automatic Router Configuration Protocol - a method by which Routers may be automatically configured according to a predefined or default scheme of behavior and address allocation.

C

CMCP

An abbreviation for ConMon Configuration Protocol (used in the context of firewall port management).

ConMon

The service that manages control and monitoring of Dante devices.

D

DBCP

DataBase Connection Pool - a cache of database connections maintained so that the connections can be reused when future requests to the database are required.

I

ISV

Independent Software Vendor

M

mDNS

The multicast DNS (mDNS) protocol resolves hostnames to IP addresses within small networks that do not include a local name server. It is a zero-configuration service, using essentially the same programming interfaces, packet formats and operating semantics as unicast Domain Name Service (DNS).

P

PTP

Precision Time Protocol - a protocol used to synchronize clocks throughout a computer network.

R

Rx

Receive

T

TCP

Transmission Control Protocol - a communications standard that enables application programs and computing devices to exchange messages over a network.

Tx

Transmit

U

UDP

User Datagram Protocol - a communications protocol that facilitates the exchange of messages between computing devices in a network.

Index

A

Activation 20, 23
Adding DAL to Your Application 19
API 8
API Access Token 9
Application Packaging and Installation 19
Applications 10
Audio API 8
Audio Transfer 24
Authenticating DAL Binaries on Windows 40
Available Channels 26

B

Building and Running the DAL Examples in Xcode 32

C

C# Examples 14
C++ Examples 11
Child Processes 22
Cleaning up Subscriptions 26
Compiling 19
Connection API 25
Connection API vs Embedded Dante API (eDAPI) 25
Connection Management 24
Connections 26
Connections API 8

D

Dal Loopback 15
Dal Loopback App 18
Dal Managed Library 14
Dal Wave Player 16
Dal Wave Recorder 17
DalConnectionsTest 13
DalLoopbackTest 12
DalWavePlay 12
DalWaveRecord 12
DanteApplicationLibrary package 9

E

Embedded Dante API 25
Events 27
Example DAL Applications 10

F

Features 7
Firewall Management (Windows) 31

G

Getting Started 9
Glossary 42

I

Installation 19
Instance API 8
Instance Management 22

L

Linking 19
Load Disallowed DAL library by System Policy on macOS 38

N

Notarize the DalLoopback Example Application on macOS 34

O

Operational Overview 25
Overview 7

P

Persistence 26
Preparation 19

R

Running 20

S

Server Socket Management 30
Specific Behaviours 28

W

Windows C# Examples 14