# Introduction

The OCAMicro reference implementation is provided by the Alliance as a lightweight open-source reference design for an embedded implementation of an OCA device. The SVN repository is accessible by all Alliance members, those that have requested access, and can also check-in additions / feature changes that may be required in the future. To anyone outside the Alliance, it is the source code is provided as a zipped export of the project, via the Alliance web site.

The implementation supports two hardware platforms, the "OCA Micro" hardware platform, used in the Alliance trade show booth and also a basic implementation that runs natively on the Audinate Brooklyn module, which is only of benefit to Dante licensees. Additionally, there are Microsoft Visual Studio 2012 projects provided with the source that allow implementations to be run on a Windows computer.

# Prerequisites

## PC Applications

Since a Visual Studio solution and projects are provided with the source, it is recommended this is installed. For running and debugging of the Windows projects, Visual Studio 2012 is recommended. This can be downloaded from here:

<http://www.microsoft.com/en-us/download/details.aspx?id=39305>

Additionally, in order for the Windows applications to run correctly, Apples Bonjour (or a compatible equivalent) should be installed on the target computer.

For Dante licensees that may want to run the Dante adaptation on their Windows host, a copy of Dante Virtual Soundcard is required, and Dante Controller may be found to be useful. Additionally, the Dante API must also be installed on the host computer. The API must be referenced in the Visual Studio **OCALiteDante** project to enable correct compilation, specifically:

* C/C++ Project Settings: the last entry under "Additional Include Directories" must be changed to point to the Dante API "Include" directory.
* Linker Input Settings: the last entry under "Additional Dependencies" must be changed to point to the Dante API library "dapid.lib", or "dapi.lib".

## STM Applications

The STM application currently only supports the "OCA PI" demonstration embedded control board that is used in the booth system. Therefore, a suitable ARM Coretex-M3 build environment is required to build this embedded application. There are two environments supported:

1. Rowley Crossworks: this was used by Attero Tech for their development of the core firmware.
2. A free GCC based environment adaption to above. This allows anyone to setup a free development system and build the embedded code.

The free development environment has a multitude of options, but the list of sources for the various components has been verified as working:

* GCC Tools: <https://launchpad.net/gcc-arm-embedded>
* Make utility: <http://liquidtelecom.dl.sourceforge.net/project/gnuwin32/make/3.81/make-3.81-bin.zip>

<http://liquidtelecom.dl.sourceforge.net/project/gnuwin32/make/3.81/make-3.81-dep.zip>

* SED utility: <http://sed.sourceforge.net/grabbag/ssed/sed-3.59.zip>

The free build environment requires the environment variable OCA\_HOME to be set:

**OCA\_HOME** - the base directory, OCAMicro Base

Once this is setup, the build can be initiated from OCAMicro/Src with "*make stm32Release*". A batch file, "*buildstm.bat*" is provided in OCAMicro that sets up the variable and starts the build.

An additional batch file, "*burn\_release.bat*" is also provided. If the user is using a Segger JLink JTAG probe and has the Segger JLink ARM tools installed, running this batch file will burn the built binary onto a connected processor.

Serial Debug Interface

CN3 on the PI PCB provides access to a UART on the STM processor, which is used for displaying debug information on a terminal. The signal levels are LVCMOS, so will need to be buffered and level shifted to connect to a traditional RS232 port. However, there are plenty of LVCMOS serial-USB interfaces available that will connect directly to this port. The serial port also supports limited input commands, specifically:

* getmac<CR>: displays the device MAC address, and
* setmac aa:bb:cc:dd:ee:ff<CR>: will set the MAC to the new specified [hex] address.

### JTAG Hardware Debugging

J1 on the PI PCB provides connection to a JTAG debugger / programmer. The connector footprint is designed for the [Tag-Connect](http://www.tag-connect.com/) debug adapters. The exact type of connector required will depend on the JTAG equipment being used, [the solutions](http://www.tag-connect.com/solutions) page may well be helpful.

## Brooklyn Applications

In order to build a native Brooklyn module application the developer will need the Brooklyn Binary Build Environment (BBE). This should be familiar to existing Brooklyn developers and new developers are encouraged to carefully study the Audinate documentation on the BBE before continuing. The build environment requires two environment variables to be setup:

**OCA\_HOME** - the base directory, OCAMicro Base

**DANTE\_PATH** - the path to the root of the BBE installation, usually in the user's home directory

There is a script file, ***buildmicro***, which sets these up with nominal values, which may be modified locally as required. Once the variables are set up, this script then calls the actual builder, ***buildbin***, which is located in Src\app\OCA\_BKNII; this script can actually be called discretely, since it sets up the environment variables with nominal values again, relative to the directory it resides in. This script calls make twice, firstly to build the library using the makefile ***makefileLIB*** and then to build the application, using ***makefileBIN***. These files should be modified to add either new library or application files.

# Visual Studio Projects

Once the solution is opened you will be presented with the projects included in the solution:

* OCALite
* OCALiteDante
* OCALiteLib
* OCAMicroBrk

The following sections describe these projects.

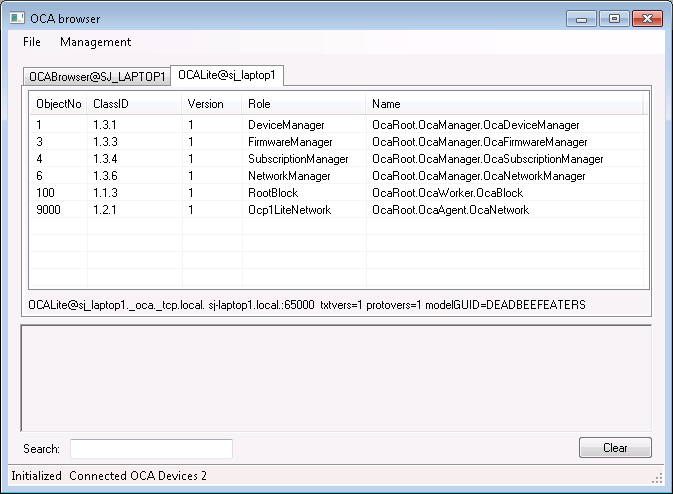
## OCALiteLib

This project is a Visual Studio librarian for the OCALite and OCALiteDante projects. It provides all the base classes for OCP.1, OCF and OCC. The project is quite well organized and it is recommended that it is browsed to see what is available. Of particular note should be the OCC tree which, at its highest level is divided into two branches, ControlDataTypes and ControlClasses. Looking at these will show the reader exactly what classes have been implemented, at that time. These will be covered in more detail in a later section.

It is not intended to delve into the inner workings of system model. Suffice to say, its structure closely follows the OCA / AES70 specification model. The reader is free to delve into this area as they wish.

## OCALite

This project is the minimal implementation of an OCA device running on a Windows computer. When run, it will appear as below in the OCA Browser (The OCA Browser is available for OCA Alliance members on the members portal):

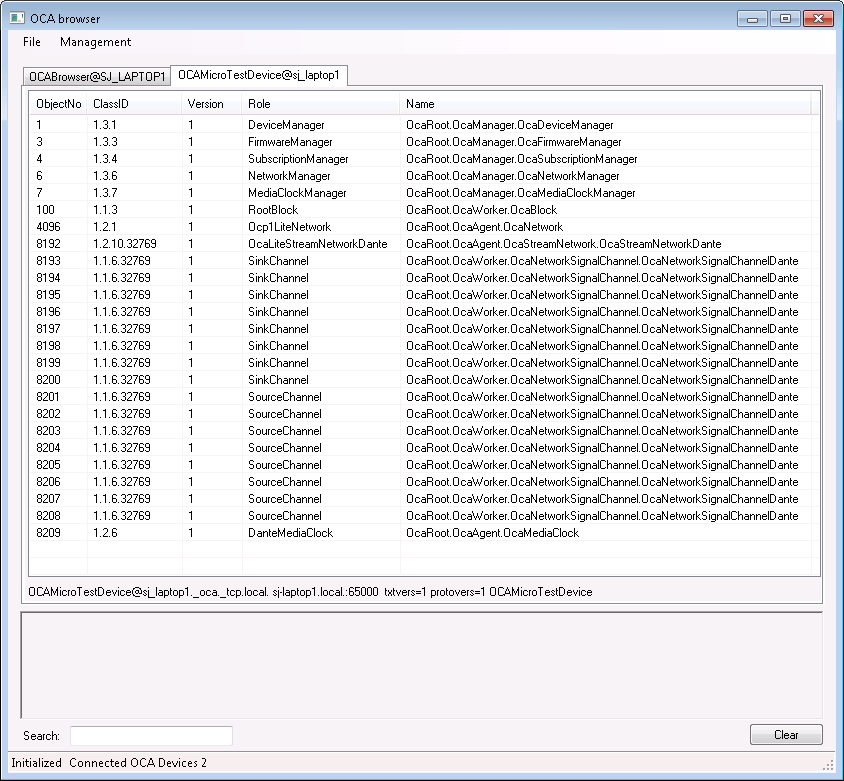


This project is a good starting point for anyone who doesn't want the Dante specific adaptation but wants to start creating an OCA device. The main() function is located in:

Src\app\OCALite\OCALite.cpp

## OCALiteDante

This project extends OCALite by adding the most basic Dante functional adaptation. Note, apart from those previously mentioned, you **must have Dante Virtual Soundcard (DVS) installed and running** on the development computer for this project to run successfully. This is so that the OcaStreamNetworkDante object can attach to Dante I/O streams on the local host because, as it's the Dante adaptation, it expects there to be audio streams present. When run, it will appear in the OCA Browser as below (The OCA Browser is available for OCA Alliance members on the members portal):



As can be seen there are more OCA objects within the device than the OCALite application, these are the additional basic Dante adaptation objects. The base Dante objects created are:

* OcaStreamNetworkDante
* OcaMediaClock
* OcaNetworkSignalChannelDante

The OcaNetworkSignalChannelDante object(s) in the above list are created by the OcaStreamNetworkDante class, the I/O channel counts being automatically parsed by the class via the Dante API to create the correct number of objects. Each of these objects can be inspected in the Browser by double clicking on them.

## OCAMicroBrk

This is not a buildable project, but a container for the Brooklyn implementation of OCALiteDante. Because it is targeted for the Brooklyn module, it can only be built from the Brooklyn binary build environment, or BBE. A brief description of the build environment for this project is given.

As with the other projects, there is a library containing the base OCP.1, OCF and OCC classes and then an application module is built that uses the library. A script is provided to set up two environment variables that the makefiles use:

**OCA\_HOME** - the base directory, OCAMicro Base

**DANTE\_PATH** - the path to the root of the BBE installation, usually in the user's home directory

There is a script file, **buildmicro**, which sets these up with nominal values, which may be modified locally as required. Once the variables are set up, this script then calls the actual builder, **buildbin**, which is located in Src\app\OCA\_BKNII; this script can actually be called discretely, since it sets up the environment variables with nominal values again, relative to the directory it resides in. This script calls make twice, firstly to build the library using the makefile **makefileLIB** and then to build the application, using **makefileBIN**. These files should be modified to add either new library or application files.

# OCA-Dante Interface

This section describes how the Dante API control and subscription interface is managed and interfaces with the OCA system.

In its most basic form, the OCA-Dante API interface is required to allow the OCA system to:

1. Discover the devices clock support abilities, specifically supported sample rates.
2. Discover the devices I/O channel configuration, and the current state of channel subscriptions.
3. Monitor changes to clock and sample rate.
4. Monitor changes to subscriptions.
5. Process OCA-side changes to clocking.
6. Process OCA-side changes to subscriptions.

In order to do this, the interface needs to provide the following functionality:

1. Ability to receive monitor and send control messages to a Dante device.
2. Ability to receive monitor and change device routing.

This functionality is provide in these files:

* DanteLiteHostInterface.cpp
* DanteLiteConMon.cpp

both of which reside in Src\app\OCALiteDante. Initialization of the Dante API must be done before the main OCA system is started, so that the devices' configuration is obtained before creating the OCA device. The core initialization is done in DanteLiteHostInterface/DanteLiteHostInterfaceInitialize(), called from main() in OCALiteDante.cpp.

Once this is completed, the system can obtain Dante specific parameters, such as the "friendly" device name, which is used for the name of the OCA device. This name is also used to initialize the control and monitoring system, DanteLiteConMon/DanteLiteHostInterfaceInitializeConMon(...).

## Remote Control

At this point, it's worth noting that the control and monitoring system is design to be able to connect to a **remote network device**. This has powerful potential for development and debugging, since the ConMon can connect and control a remote Dante device from an application running and / or being debugged on a Windows computer. This model is very efficient for developing applications that will ultimately run on a Brooklyn module, enabling very quick compile / debug cycles, especially compared to doing this on the actual Brooklyn module itself. The routing side of the interface will now attach to either a local or remote host, so the need to always have DVS running even when connecting to a remote Brooklyn module has been removed. In this case, all the Dante objects are created with respect to the capabilities of the remote target Brooklyn module.

## Dante Class Adaptations

There are three Dante adaptations from the standard OCA classes required in this system:

1. OcaStreamNetworkDante, from OcaStreamNetwork, Src\app\OCALiteDante\OcaLiteStreamNetwork.cpp
2. OcaNetworkSignalChannelDante, from OcaNetworkSignalChannel, Src\app\OCALiteDante\OcaLiteNetworkSignalChannelDante.cpp.
3. OcaMediaClock, from Src\app\OCALiteDante\OcaLiteDanteMediaClock.cpp.

The latter isn't strictly an adaptation, but it is specific to Dante networks, so is included here.

The Dante API needs to interact with the OCA objects created, specifically the instance of the OcaStreamNetworkDante class and the OcaMediaClock class. This achieved by the classes in their initialization phase by calling the self-explanatory interface functions:

* DanteLiteHostInterfaceSetStreamNetworkObject(), and
* DanteLiteHostInterfaceSetMediaClockObjectNumber()

with their respective [unique] object numbers. It is anticipated that other future adaptations would use the same method.

These adaptation classes have been created to give the minimal generic functionality considered necessary. Consequently, some features may not be available. An example of this is the OcaMediaClock. At the time of writing, it correctly parses the device's supported sample rates and is able to change the device's sample rate, but it does not attempt to parse the capabilities of, or control the external clock input, since this would be highly device specific.

## Monitoring Interface

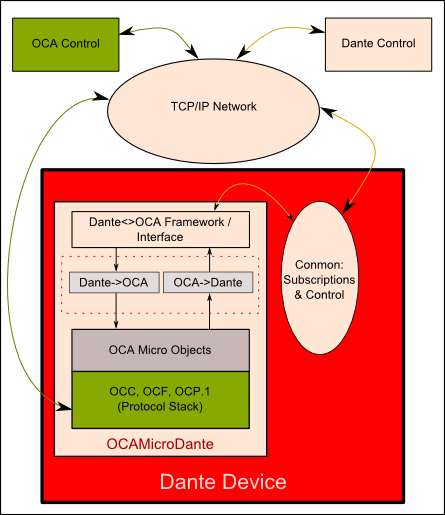
The Dante API monitoring interface is handled by call-back message handlers in DanteLiteConMon.cpp. Most of these monitoring messages are processed by the function **DanteCMCbMonitoring(...)**. This function inspects the messages for their origin Vendor ID (VID), and will automatically pass any messages with the Audinate VID to the function **DanteLiteStatusMessageCallback(...)**. This function captures all the messages that are of interest to support the Dante adaptation classes. This may well be expanded in the future to support further adaptations, or enhancements of current functionality. Once the Audinate VID message has been processed by the adaptation handler the message is passed onto an optional function handler, defined by a function pointer, *pDanteCMCbMonitoringVendor*. This function pointer is set at initialization by **DanteLiteHostInterfaceInitializeConMon(...)**. There are actually two function pointers, the other being pDanteCMCbMeteringVendor, which is used for device metering call-backs, initialized at the same time. The initialization values of these function pointers are determined by two functions:

1. GetDeviceStatusMessageCB(), and
2. GetDeviceMeteringMessageCB().

Both of these functions are defined in OCALiteDante.cpp, with the intention being that these are changed by the user to point to vendor specific functions handlers, rather than returning *NULL* as they currently do. This allows user-specific code outside the remit of the OCA Micro base to be able to intercept any VID sourced monitoring messages, and also separately process metering messages from the device. This gives huge flexibility for message handling in any implementation, as may be required.

## Communication Model

The diagram below shows how the OCA Micro Dante Interface fits into the device and network:



# OCC Supported Classes

This is a list of the currently supported OCA classes in the OCAMicro implementation.