Barco Overture

Control Server Driver SDK

1 Overview

This document describes how to write plugins such as drivers and behaviors for OvertureCS.

OvertureCS is a "Control Server" which:

- monitors and controls real-world devices via 'Device Drivers'.
- implements domain specific logic via 'Behaviors'
- communicates with the rest of Overture.

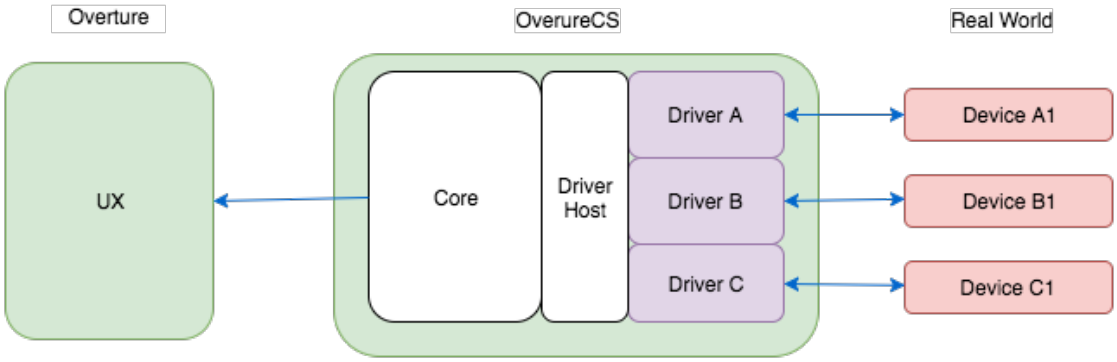
Note that although 'Device Drivers' and 'Behaviors' serve different purposes they are implemented using the same technology and architecture. In practice a 'Behavior' differs from a 'Driver' only by the fact it usually doesn't interact with a real world device but rather with other entities defined in Overture.

The term 'Driver' will be used in this document for both 'Device Drivers' and 'Behaviors'. See Behaviors for behavior specific details.

1.1 OvertureCS Architecture

OvertureCS is made of:

- Core: the OvertureCS engine component which manages communication with the UX Server
- Driver Host: a component which hosts drivers and provides an API between the drivers and the core
- Drivers: a set of specific drivers which communicate with real world devices



Overture users install the driver plugins using the Overture Configurator and define the list of devices which must be controlled by a specific OvertureCS in the Overture Configurator.

2 Quick Start

This section is a short walk-through which shows how to:

- Write a simple driver
- Package the driver in the zip folder
- Install the driver in Overture UX Server
- Create a point of type device which uses this driver

2.1 Creating a simple driver

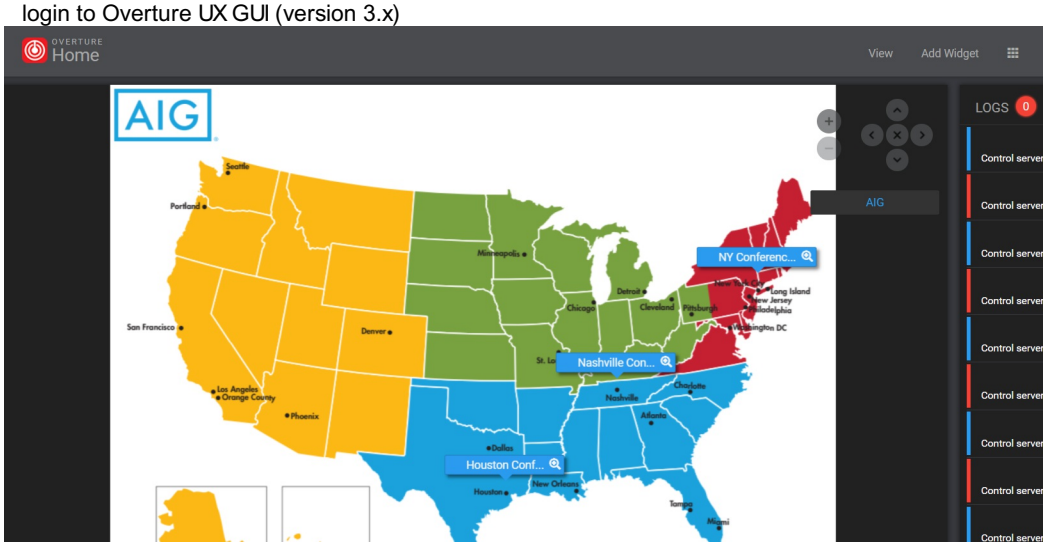
- create a folder named `overture_quickstart`
- open this folder in your text editor
- create a file named `package.json`
- copy the content of the section Quickstart Example: package.json into this file.
- create a file named `index.js`
- copy the content of the section Quickstart Example: index.js into this file.
- create a file named `readme.md`
- copy the content of the section Quickstart Example: readme.md into this file.

2.2 Packaging the driver files

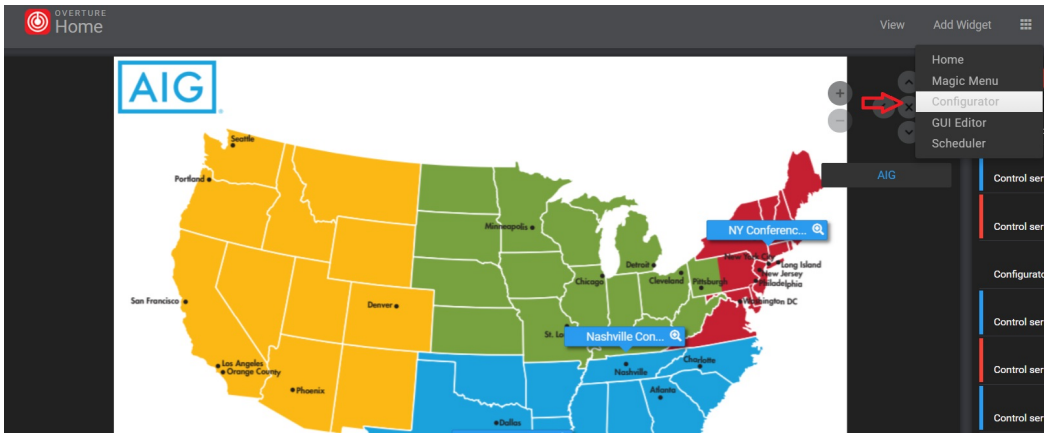
- select `package.json`, `index.js` and `readme.md` files and compress them into a zip file
- rename the zip file as `overture_quickstart.0.0.1.zip`

2.3 Installing the driver in Overture UX

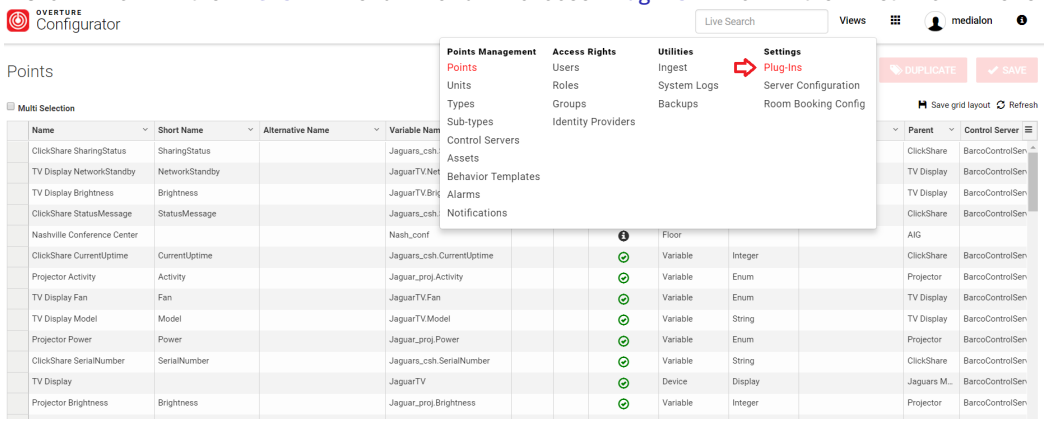
- login to Overture UX GUI (version 3.x)



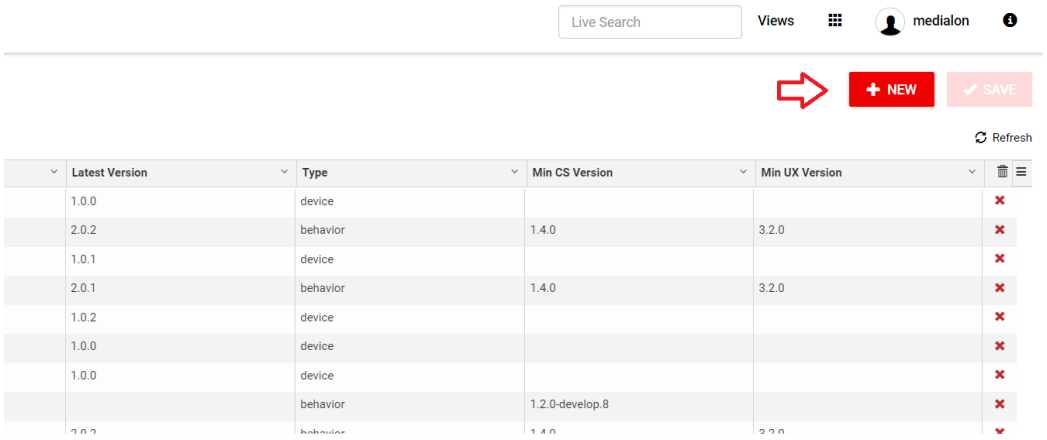
- click on the square list menu icon at the top right corner of the screen
- choose Configurator from the list of Apps



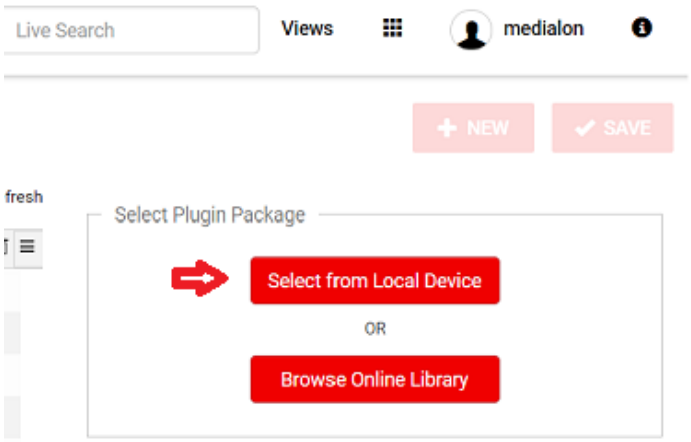
- Click on the `Views` menu and choose `Plugins` from the list of views



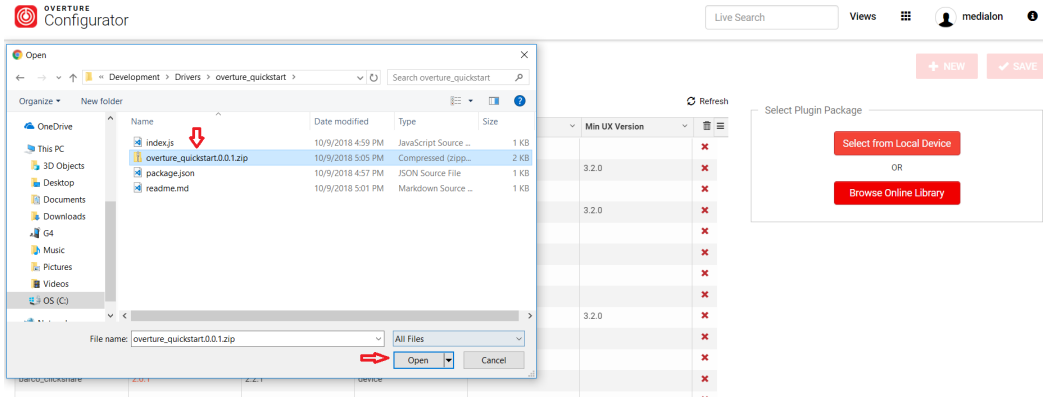
- at the top right of the screen click `New` button



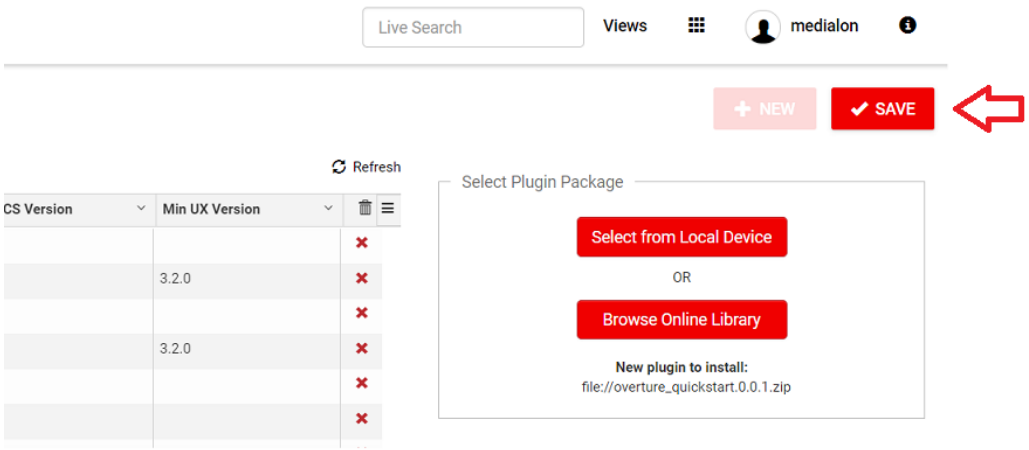
- Two buttons will apear on the screen, click on the `Select from Local Device` button



- browse to the zip file which you created earlier (`overture_quickstart.0.0.1.zip`) and click "open" button in the browsing window



- Click on the "Save" button at the top right corner of the screen

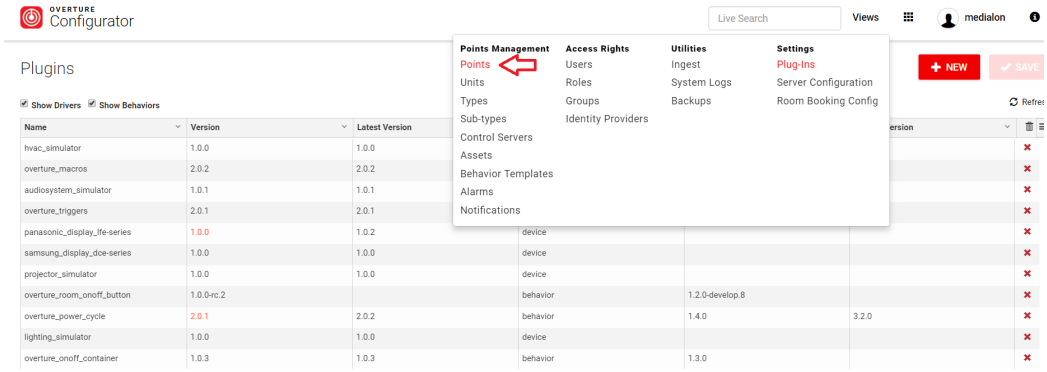


- the driver is installed in Overture UX

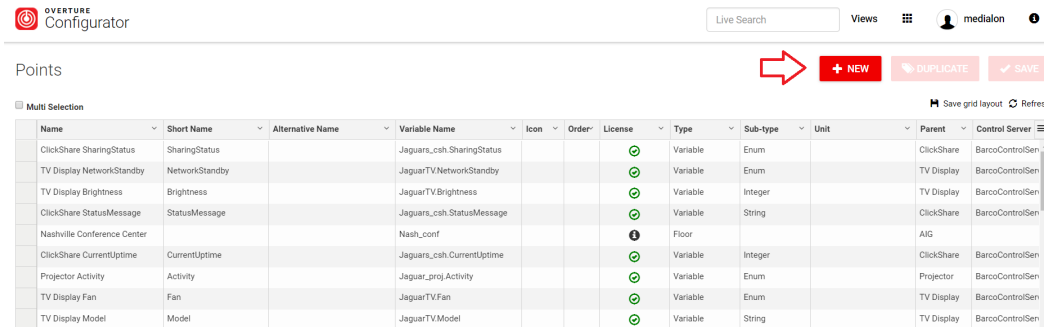
2.4 Creating a device

- login into Overture UX GUI (version 3.x)

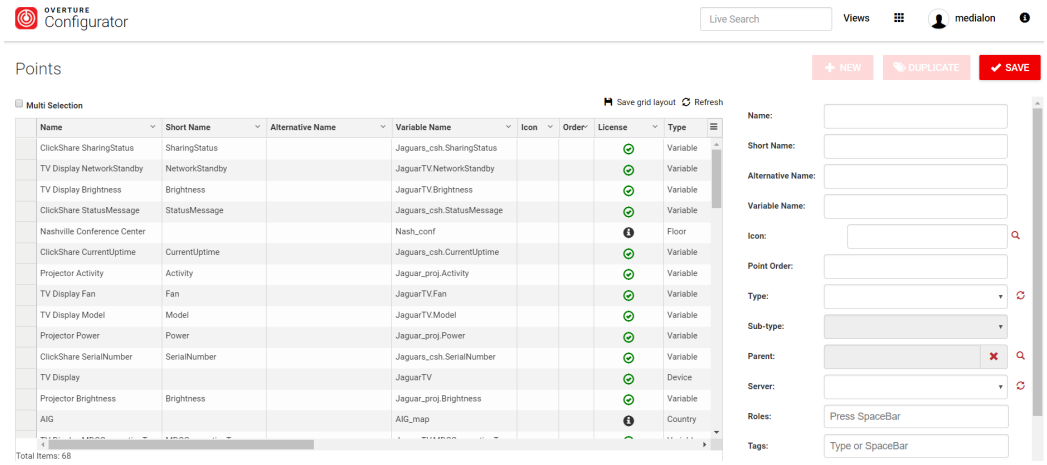
- click on the square list menu icon at the top right corner of the screen
- choose Configurator from the list of Apps
- the `Points` page should be displayed by default (if you are on any other page, then go to `Views` menu and choose `Points` from the list of views)



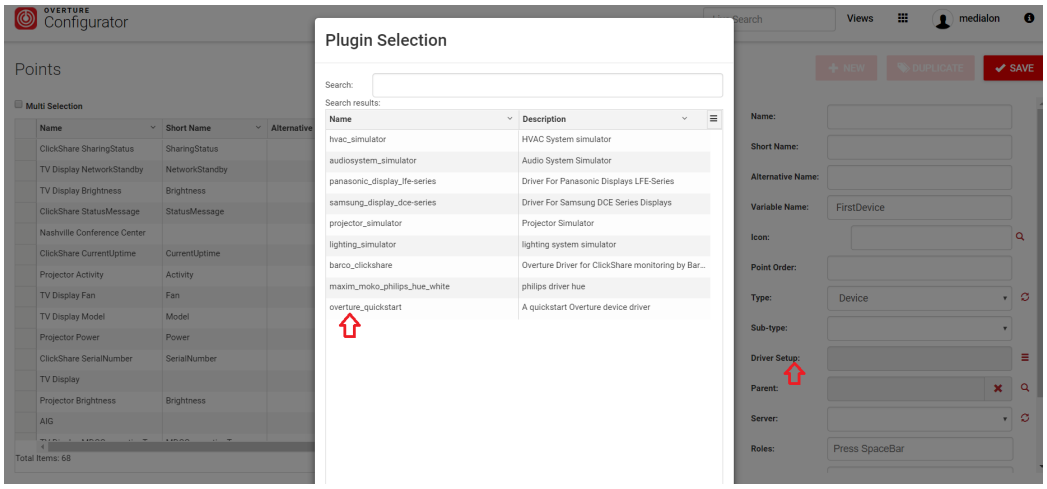
- Click on the `New` button at the top right corner of the screen



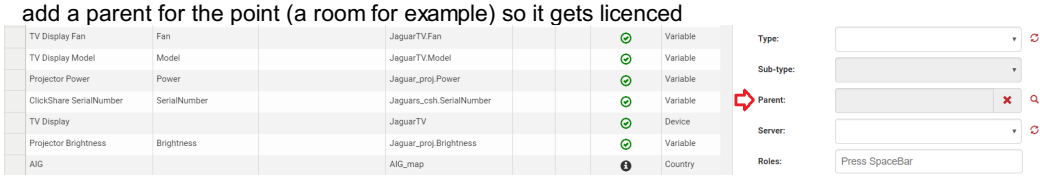
- The point property form will appear on the right side of the screen



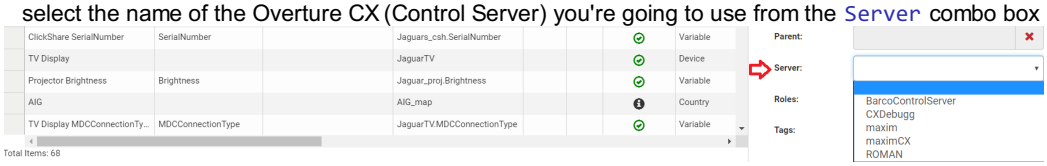
- set 'Variable Name' as `FirstDevice`
- set 'Type' as `Device` (after you set the Type, `Driver Setup` field will appear down below)
- click on the `Driver Setup` field and select `Overture Quickstart` in the driver Plugin dialog, and confirm your choice by pressing the `ok` button



- add a parent for the point (a room for example) so it gets licenced



- select the name of the Overture CX (Control Server) you're going to use from the `Server` combo box



- click the `Save` button at the top right of the screen
- the device point is created along with 2 variables: `Activity` and `Value` (you can see them in the point's view below the device point)

2.5 Starting Overture Control Server

- Download and install Barco-OvertureCS-XXX-Setup.exe at Overture Downloads (Barco account required)
- The OvertureCS will be automatically started as a service.
- Check that your Control Server is Connected to Overture UX (on the GUI of the Control Server, the Status should display Connected).

If it is not connected, you might check that the Control Server is properly configured in the Configurator, and that your Control Server is contacting the correct Overture UX (usually `http://localhost`).

- login to Overture UX GUI (version 3.x)
- click on the square list menu icon at the top right corner of the screen
- choose Configurator from the list of Apps
- Click on the `Views` menu and choose `Control Servers` from the list of views

2.6 Using the device

That's it! Now, you can use this device as other installed device in Overture. OvertureCS provides a user

interface where you can perform basic diagnostics and tests on this driver.

3 Files

A driver is at least made of the following mandotory files:

- `package.json`: a JSON file which describes the driver and its properties
- `index.js`: the main JavaScript file which implements the driver behavior
- `readme.md`: a Markdown readme

Other files may be added by the driver author depending on the requirements of the driver. For example: `node_modules` or any files required by the driver code for the specific driver implementation.

3.1 Package JSON

A standard NPM `package.json` manifest file describes OvertureCS drivers. All Overture specific information are stored under the `overture` property.

```
{
  "name": "overture_generic_projector",
  "version": "1.0.2",
  "description": "A generic projector",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Barco",
  "license": "SEE LICENSE IN license.txt",
  "devDependencies": {},
  "dependencies": {},
  "overture":{
    "brand": "Overture",
    "models": ["Name of the series", "Name of a specific supported model"],
    "type": "device",
    "subtype": "projector",
    "pointSetupSchema": {},
    "variables": []
  }
}
```

3.1.1 Standard package.json keys

OvertureCS requires the following keys to be defined and enforces their format:

- `name`: the name of the driver. The name must follow the `<brand>_<model>_<type>` pattern. Allowed chars:
  - alpha-numerical characters
  - underscores `_`
  - dashes `-`.
  - regular expression format: `[A-Za-z][A-Za-z0-9_-]*`.

  Examples: `barco_DLPLaserPhosphor_projector` ,`barco_projector_c-series`.
- `version`: format `major-minor-patch`. Example: `1.0.2`. It should be updated whenever one of the driver source files is modified.
- `main`: the name of the main script file of the driver. Usually `index.js`.

3.1.2 overture main keys

The `overture` keys describe the Overture specific information of the driver. The main keys are:

- `overtureCSVersion`: the overtureCSVersion key is optional indicates the minimal OvertureCS version the driver expects. OvertureCS will load the driver only if its own version is equal or upper than the overtureCSVersion specified in the driver. This optional key should be specified if the driver uses a new functionality provided by the OvertureCS. For example, a driver using createVariable which is available since OvertureCS version 1.2.0 should have a overtureCSVersion equals to 1.2.0 in its package.json.
- `brand`: the brand of the device which is controlled by the driver. The brand `overture` must be used for drivers which are not related to a specific brand and for 'Behaviors'.
- `models`: the models (or model family) of the device which can be controlled by the driver. Can be an empty array.
- `type`: MUST BE `device` for Overture 'Device Drivers' or `behavior` for Overture 'Behaviors'.
- `subtype`: the subtype of the device (can be ommitted for behaviors). The subtype must match pre-defined Overture device subtypes:
  - `generic`
  - `lighting`
  - `hvac`
  - `power`
  - `camera`
  - `audiosystem`
  - `projector` (Video projector)
  - `avconference` (Audio/Video Conference)
  - `matrix`
  - `display`
  - `lift`
  - `shades`
  - `player` (Video player)
  - `io`
- `pointSetupSchema`: see overture.pointSetupSchema.
- `variables`: see overture.variables.
- `commands`: see overture.commands.

3.1.3 overture.pointSetupSchema

This key describes the JSON Schema for the device setup object. Uxserver uses it in order to automatically generate the setup dialog for a device created by the driver (see picture).

The following example generates the matching setup dialog and populates the following `config` object which is passed to a device `setup()` function when the driver instantiates the device.

Schema:

```
{
  "pointSetupSchema": {
    "title": "Generic Projector",
    "type": "object",
    "properties": {
      "host": {
        "type": "string",
        "description": "Host address",
        "default": "192.168.1.10"
      },
      "port": {
        "type": "integer",
        "default": 4352
      },
      "auth": {
        "type": "boolean",
        "format": "checkbox"
      },
      "password": {
        "type": "string"
      }
    },
    "required": ["host", "port"]
  }
}
```

Setup Dialog:

Config Object generated:

```
{
  "host": "192.168.1.10",
  "port": 4532,
  "auth": false,
  "password": ""
}
```

IMPORTANT: due to a current limitation of the Driver SDK, the JSON Schema must not describe a property of name `id` at the root of the config. For example, the following schema is not valid:

```
{
  "pointSetupSchema": {
    "title": "Generic Projector",
    "type": "object",
    "properties": {
      "id": {  // WRONG! Cannot use 'id' as a property of the schema
        "type": "string"
      }
    },
    "required": ["id"]
  }
}
```

Some resources about JSON Schema:

- JSONSchema.net: allows creating a schema from a JSON example
- JSON Editor: shows the result dialog given a JSON schema
- JSON Schema Examples: gives some JSON Schema examples

3.1.4 overture.variables

This key describes the static variables exposed by the driver. This is an array of `variable` objects. Note: "dynamic" variables are not described by this key but are instead created at runtime in the driver device's `setup()` function.

```
{
  "variables": [
    { "name":"Status", "type":"enum",
      "enums":["Disconnected","Connected"]
    },
    { "name":"Power", "type":"enum",
      "enums":["Off","On"],
      "perform" :{ "action":"Set Power", "params":{ "Status":"$string"}}
    },
    { "name":"Brightness", "type":"integer",
      "min":0, "max":100,
      "perform" : { "action":"Set Brightness", "params":{ "Level":"$value"}}
    },
    { "name":"Temperature", "type":"integer" }
  ]
}
```

A `variable` object has the following common properties:

- `name`: the name of the variable. Convention is to capitalize the name.
- `type`: the type of the variable. Can be:

        - `string`
        - `integer`
        - `enum`
        - `real`
        - `time`
        - `date`
    - `perform`: (optional) the command associated to the variable. See variable.perform
    - `smooth`: (optinal) whether the variable change should be 'smoothed'.
        - If `smooth` is `true`: the variable change will be smoothed using the default smoothing delay (1000 milliseconds).
        - If `smooth` is a number in milliseconds: this value will be used as the smoothing delay for this variable. See Smoothing Variable Changes
    - `readonly`: (optional) whether the variable is read-only (`true`) or writeable (`false`). Defaults to `false` if there's a `perform` key.
    - `unit`: (optional) the unit the variable should be displayed with. The `unit` key must match the name of one of the units defined in the UX.
        - `celsius`: Celsius [°C]
        - `fahrenheit`: Fahrenheit [°F]
        - `leveldB`: Level dB [dB]
        - `unitsperhour`: Units per hour (/hr) [/hr]
        - `waterpH`: Water pH [pH]
        - `pressurepa`: Pressure Pa
        - `pressurepsi`: Pressure PSI
        - `pressurebars`: Pressure Bars
        - `rateofflowm3s`: Rate of flow m3/s [m3/s]
        - `rateofflowf3s`: Rate of flow f3/s [f3/s]
        - `rateofflowlmnlpm`: Rate of flow l/mn (LPM) [l/mn]
        - `rateofflowgmgpm`: Rate of flow g/m (GPM) [g/m]
        - `relativehumiditypercent`: Relative Humidity (%) [%]
        - `speedmhmilesperhour`: Speed m/h (miles per hour) [MPH]
        - `speedmsmeterspersecond`: Speed m/s (meters per second) [m/s]
        - `speedrpmrevolutionsperminute`: Speed RPM (revolutions per minute) [RPM]
        - `speedkmhkilometersperhour`: Speed km/h (kilometers per hour) [km/h]
        - `levelpercent`: Level % [%]
        - `hours`: Hours [hrs]
        - `datefulldate`: Date [fullDate]
        - `datelongdate`: Date [longDate]
        - `datemediumdate`: Date [mediumDate]
        - `dateshortdate`: Date [shortDate]
        - `timemediumtime`: Time [mediumTime]
        - `timeshorttime`: Time [shortTime]
        - `watts`: Watts [W]
        - `kwh`: kW/h [kW/h]
        - `volts`: Volts [V]
        - `amps`: Amperes [A]
    - `icon`: (optional) the icon the variable should be displayed with. The `icon` key must match the name of one of the icons available in the UX.
    - `widget`: (optional) specifies the widget which should represent this variable in the user interface provided by UX. Example: `med-slider` for variable which is best represented by a slider.

Each type of variable also has specific properties:

- for type `integer`:
    - `min`: the minimal value of the variable. Default taken by the clients is `0`.

        - `max`: the maximal value of the variable. Default taken by the clients is `100`.
    - for type `enum`:
        - `enums`: an array of strings which describes the various enum values of the variable
    - for type `real`:
        - `min`: the minimal value of the variable. Default taken by the clients is `0`.
        - `max`: the maximal value of the variable. Default taken by the clients is `100`.
        - `precision`: the number of digits after the decimal point. Default is `2`.

3.1.4.1 variable.perform

OvertureCS uses the `variable.perform` key in order to map variable change requests to the matching device commands. This allows to easily implement 'intention programming' where the user programs the system mostly by setting the required values of variables instead of sending commands.

Perform keys:

- `action`: the name of the device command associated to the variable
- `params`: the parameter(s) of this command. The params can make use of 2 macros:
    - `$value`: the required value of the variable
    - `$string`: the required value translated to a string (useful for enums variable where the value `1` can be translated to the string `Night Mode` for example).

As an example: the following `perform` key associates the `Level` variable to the command `Set Level($value)`.

```
{
  "name": "Level",
  "type": "integer",
  "perform": { "action":"Set Level", "params": { "Level" : "$value"}}
}
```

In this case, OvertureCS translates

- `setVariable('Server.LightSystem.Level', 20)` request to:
- `perform('Server.LightSystem', 'Set Level', { Level: 20})`

3.1.4.2 Smoothing Variable Changes

Smoothing variable changes is related to `variable.perform` is particularly useful when a variable is linked to a slider. The smoothing algorithm avoids 'jumpy' behavior of sliders (or other similar GUI widgets) by implementing an 'optimistic change behavior'.

In practice, this means that the driver will pretend that a variable changes 'immediately' as soon as the new required value is received and BEFORE receiving a feedback from the real world device.

The variable value is updated when the real world device sends back a response eventually (and if no other commands has been sent in the meantime). The variable value is reset if there's no response from the real world device. The detail of the algorithm will not be covered in this document.

3.1.5 overture.commands

The `overture.commands` key describes the static commands exposed by the driver. The `commands` key is an array of `command` objects.

```
{
  "commands" : [
    {
      "name": "Set Level",
      "params": [
        { "name": "Level", "type": "integer" }
      ]
    }
    // other commands...
  ]
}
```

Keys:

- `name`: the name of the command (as seen by the user)
- `params`: the parameters of the command
- `optimize`: (optional) specifies whether the Command Manager should optimize the queuing process for this command. See Optimizing Command Queuing
- `alias`: specifies the real function name of the device associated to this command. This properties is useful when the desired command name clashes with one of the standard SDK device function. Example: if `alias` is set to `setStop` for a command whose name is `Stop`: the device `setStop()` function will be executed when a `perform('Stop')` is sent to the device.

Name conversion: when OvertureCS receives a command request, it tries to call a matching function in the driver device using the following algorithm:

- use command name 'as is'. ex: `device['Set Level'](params)`.
- then use camel-cased version the command name. ex: `device.setLevel(params)`

3.1.5.1 Parameter format

The `command.params` key is an array of parameter objects. The parameter object format depends on the type of the parameter:

- integer: `min` and `max` are optional.

```
{ "name":"Value", "type": "integer", "min": 0, "max": 100 }
```

- string:

```
{ "name":"Value", "type": "string"}
```

- enum:

```
{ "name":"Mode", "enums": ["Off","On"]}
```

- real: `min` and `max` and `precision` are optional.

```
{ "name":"Value", "type": "real", "min": 0, "max": 100, "precision": 2 }
```

- time:

```
{ "name":"Value", "type": "time" }
```

- date:

```
{ "name":"Value", "type": "date" }
```

3.2 index.js

The `index.js` file is the main entry file of the driver. This file is a NodeJS module written in JavaScript. OvertureCS versions under 1.4.0 used NodeJS 6. OvertureCS 1.4.0 uses NodeJS 8.

Supported JavaScript version:

- `ES2015`
- `ES5`
- `TypeScript` (transpiled to either `ES2015` or `ES5`)

Note that the code examples of this documents are written in ES2015 but can be easily back ported to ES5.

The overall structure of `index.js` is:

- an import section which refers to NPM modules (if the driver has dependencies)
- an export section of driver standard functions as expected by the Driver SDK
- a device object factory which creates an object with the following sub-sections:
  - standard device functions as expected by the Driver SDK
  - a set of device specific published commands
  - other internal functions needed for the driver implementation

General Layout:

```
// (optionaly) import NPM dependencies
let request = require('an-npm-module')

// required SDK init function
exports.init =  function (host) { /* ... */ }

// required SDK createDevice function
exports.createDevice = function (base) {

  // standard SDK device functions
  function setup(config) { /* ... */ }
  function start() { /* ... */ }
  function stop() { /* ... */ }
  function tick() { /* ... */ }

  // device functions for published commands
  function setPower(params)  { /* ... */ }

  // internal (private) device functions
  function aPrivateFunction() { /* ... */ }

  // return an object which exposes the public device functions
  return { setup, start, stop, tick, setPower }
}
```

The inner details of implementing a driver is described in the Writing a Driver section.

3.3 readme.md

The `readme.md` file is the standard 'read me' file usually provided with NPM modules. This file should describe the purpose of the driver and how to use it. Its content is automatically displayed when the user browses the Overture plugins repository or when she installs a driver.

It should have the following sections:

- Overview: an overview of the driver
- Setup: a description fo the setup dialog and its properties
- Variables: the list of the variables published by the driver
- Commands: (optional) the list of the commands published by the driver
- Revisions: the list of driver sofware revisions
- License: license information

3.4 Driver distribution package

A driver is distributed as a `.zip` compressed file which includes all of the source files of the driver (including `node_modules` NPM dependencies if any).

3.4.1 Driver package name

The `.zip` file name is derived from the driver name and version using the `<drivername>.<version>.zip` format.

For example: if the name of the driver (as described in `package.json`) is `overture_generic_projector` and the version is `1.0.2`, the driver package file must be named `overture_generic_projector.1.0.2.zip`.

3.4.2 Driver package distribution

There are two ways to install a driver package in UX:

- download and install directly from the Overture WEB site
- upload a file from the a host file system (ex: USB key)

4 Writing A Driver

This sections gets deeper into writing drivers.

4.1 Device general concepts

Overture exposes the functionalities of a driver device to client applications using 2 concepts:

- variables: Variables are the main way a driver device exposes its functionalities.
  - Client applications are automatically notified when a device variable changes.
  - Client applications can also write a value into a variable to trigger a specific device action.
  - Thus a variable has an Actual Value which represents the real world value and a Required Value which describes the state a client application wants to put the device in.
- commands: Exposing commands is useful when the "one variable Actual/Required concept" cannot fully describe a functionality of a device.

4.1.1 Static vs Dynamic variables

"static" device variables are variables which are created regardless of the setup of the device. This has to be opposed to "dynamic device variables" which are created by the driver code at runtime.

As an example, let's say a driver device always has a unique `Status` variable but can have several "ChannelX" dynamic variables depending on the result of the device setup dialog. The `Status` variable is said to be "static" whereas the "Channel1", "Channel2", etc. variables are "dynamic".

4.2 Driver Lifetime

Driver and driver device lifetimes are made of 3 phases:

- Initialization:
  - The driver is loaded by the host, the driver `init()` function is called
  - Then the host calls the driver `createDevice()` function which must return an object representing a device. The host calls `createDevice()` for each device defined in the installation project.
  - Then the host calls the `setup()` function of each device object.
- Device Runtime:
  - The host starts the device runtime phase by calling the device `start()` function
  - While the device is 'running':
    - The host repeatedly calls the device `perform()` function (or automatically translate `perform()` to device specific command functions) when a command is issued by a client application (ie Dashboard, MagicMenu or UX Behavior).
    - The host also regularly calls the device `tick()` function
  - The host terminates the device runtime phase by calling the device `stop()` function. It's VERY IMPORTANT that all the resources (sockets, memory, timers, etc.) be released by the device implemention of the `stop()` function.
- Teardown
  - The host automatically tears all of the device down when the host exits. The host also calls the driver `cleanup()` function if the driver exports this function. Note that this phase usually doesn't require any implementation at the driver source code level.

Note that the Device Runtime phase can be repeated several times depending on the current project. A device can be started and stopped several times between the Initialization phase and the teardown phase. This is dependent on the installation project and whether the device `Activity` variable is set or reset by client applications.

4.3 Driver Module Interface

The main entry point is the driver module which consists of the following exported functions:

- `init()`: intializes the driver
- `createDevice()`: asks the driver to create a device
- `cleanup()`: (optional) asks the driver to release resources

4.3.1 driver.init

`init()` is the first function of the driver called from the SDK. This function is called with the `host` parameter.

Parameters:

- `host`: an object representing the host of the driver. This object exposes several properties which allows the driver taking advantage of several services provided by the host system. This object implements the IHost interface.

It is usually a good idea to keep a reference to the `host` object in the top scope of the driver. This allows using the services provided by `host` easily in the rest of the code.

The most commonly used property of the host is `logger` which allows adding messages in the application logs:

- `logger`: an object which allows logging messages on the host logging system. This object implements the ILogger interface. Messages can be logged using the `info()`, `debug()`, etc. functions.

Check IHost interface out for other useful services.

Example of `init()` implementation:

```
let host
exports.init = function (_host) {
  host = _host // keep a ref to host for future usage
}
```

4.3.2 driver.createDevice

`createDevice()` is a function called whenever a device from this driver must be created. This function must be exported from the driver via (using `exports.createDevice=...`). The standard implementation of this function is to create an object which exposes the functions expected by the host.

```
exports.createDevice = function createDevice(base) {
  // define the device object functions

  // return an object which exposes the functions of the device
  return { setup, start, stop, tick, setPower }
}
```

see base device.

4.3.3 driver.cleanup

The optional exported function is called when the host tears the driver down. The driver MUST implement this function if system resources have been allocated by the `init()` function.

4.4 Device Object

The driver must create a 'device object' which complies with a set of simple rules. It must expose a set of functions which follow an `IDevice` interface (note: this document uses TypeScript interface definitions in order to describe JavaScript objects but using TypeScript is NOT required to develop drivers).

```
interface IDevice {
  // required functions
  setup(config :any)
  start()
  stop()

  // optional
  tick?()
  perform?(action :string, params :any, options?: IPerformOptions)
  setVariable?(variable :IVariable)
  destroy?()
  onProjectLoaded?()
}

interface IPerformOptions {
  timeout?: number
  commandType?: EPerformCommandType
}

enum EPerformCommandType {
  NORMAL = 'normal',
  EXECUTEIFIDLE = 'executeIfIdle',
  EXECUTEINPRIORIRTY = 'executeInPriority',
}
```

Additionaly, this object also implements driver specific functions which are called when a client application sends a `perform()` command to a device created by this driver.

- `destroy()`: (optional) this function is called when the device is destroyed by the ControlServer.
- `onProjectLoaded()`: (optional) this function is called after the ControlServer has received a new project from UX Server (or when it loads a project from disk).

ES2015 implementation example:

```
// driver device implementation
exports.createDevice = function(base) {
  // called by the host when the device receives its configuration
  function setup(config) {
    // ...
  }

  // called when the device starts
  function start() {
    // ...
  }

  // called when the device stops
  function stop() {
    // ...
  }

  // command (called when user perform('Set Value', { Value: ...}))
  function setValue(params) {
    // ...
  }

  // export the device public functions
  return { setup, start, stop, setValue }
}
```

IMPORTANT: The driver should not allocate system resources (module require(), sockets, files, etc.) at device creation time. System resource allocation must by postponed until the `start()` function is called.

4.4.1 Base Device

The `base` parameter is a reference to an object maintained by the host which exposes the IBaseDevice interface. The `base` device implements many low level device functionalities and chores. This allows the driver author to focus on the core implementation on the driver functionalities without having to write too much boilerplate code.

The `base` device allows a driver device to:

- Manage device variables:
  - get a reference to a device variable via `getVar()`
  - create device variables via the various `create(Type)Variable()` functions.
- Manage aysnchronous command processing: this is an advance topic covered in the Commmand Manager section.

Variables declared in the `package.json` manifest file are considered 'static' and are created on the `base` device BEFORE `createDevice()` is called. This means that static device variables can be accessed from within the `createDevice()` function.

4.5 Device Setup

After a device is created via the `createDevice()` exported function, the host calls the device's `setup()` function passing a `config` object.

The `config` object is a JavaScript version of the JSON object created by the device setup dialog in the UX Configurator Setup Page. The internal content of the `config` object is entirely up to the driver author and MUST be described in the `package.json` pointSetupSchema JSON Schema.

Example of `setup()` implementation (using the following config object):

Config object:

```
{
  "host": "192.168.1.10",
  "port": 4532,
  "channels": 2
}
```

`setup()` implementation:

```
exports.createDevice =  function(base) {
  // private variables kept inside the object closure
  let host
  let port

  function setup(config) {
    host = config.host
    port = config.port
  }

  return { setup, /* other exported function */ }
}
```

4.5.1 Dynamic Variables

As stated earlier, the driver must create the device's 'dynamic' variables in the `setup()` function. Creating a variable is done by calling the various `create(type)Variable` functions exposed by the base device. See IBaseDevice interface.

The following example shows how a driver creates as many `ChannelX` integer variables as required depending on the device configuration:

```
setup(config) {
  for (let i = 0; i < config.channels; i++) {
    base.createIntegerVariable('Channel' + i)
  }
}
```

Using the `createVariable()` function (instead of `create(type)Variable`) is recommended if there's a need to specify more metadata when creating a dynamic variable. See IBaseDevice Interface for a formal definition of the options. Below is an example which sets multiple properties to dynamic variables:

```
setup(config) {
  for (let i = 0; i < config.channels; i++) {
    base.createVariable({
      name: 'Channel+(i+1)',
      type: 'integer',
      perform: {
        action: 'Set Channel',
        params: {
          Channel: i+1,
          Level: '$value'
        }
      },
      min: -80,
      max: 20,
      unit: 'leveldB',
      icon: 'ion-android-volume-up'
    })
  }
}
```

IMPORTANT: The driver should not allocate system resources (module require(), sockets, files, etc.) at device setup time. System resource allocation must by postponed until the `start()` function is called.

4.6 Variables Management

4.6.1 base.getVar()

The `base.getVar()` function allows getting a reference to a device variable by specifying its name. `getVar()` returns a reference to an object which implements the IVariable Interface.

Note that `getVar()` returns a special empty variable object when no variable with the specified name can be found. Accessing any of the `IVariable` properties of this special variable generates a log at the `debug` level.

4.6.2 Actual Value

A driver sets the actual value of a device variable by setting the variable `value` property. It gets the actual value by getting the `value` property.

Example:

```
let status = base.getVar('Status')
status.value = 10 // sets the actual value of "Status"
logger.debug(status.value) // gets the actual value: logs "10"
```

Note: setting `.value` triggers a notification to all of the client applications (provided the value has been modified). This means that updating `.value` at a very high rate can involve a lot of network traffic and must be avoided.

4.6.3 Required Value

A driver generally doesn't need to access the required value of a static variable as the `base` object takes care of the required state management.

However, a driver DOES NEED to process the required value of dynamic variable. This can be done in the `setVariable()` function.

Example of a simplistic (not production ready) management of dynamic variables:

```
// create as many DimmerXX variables as required by the setup config
// and remember the channel mapping in the variable itself
setup(config) {
  for (let channel = 0; channel < config.channelCount; channel++){
    let variable = base.createIntegerVariable("Dimmer" + channel)
    variable.channel = channel + 1
  }
}

// implement the setVariable() method using .required
setVariable(variable) {
  socket.write(`SET DIMMER ${variable.channel} VALUE ${variable.required}\n`);
}
```

4.7 Commands

4.7.1 Device specific commands

4.7.1.1 Device specific commands overview

A driver implements device commands by merely adding a function on the device object.

A naming convention allows the base device to automatically map an external perform command to the matching device function. The convention is to use a 'camel cased' version of the external perform command as the name of the device function.

Example:

- when a client application issues an external `perform('Set Value', {Value: 100})` command
- the `base` device object tries to call a device function named `setValue()`
- the `setValue()` device function receives the `{Value: 100}` object as parameter.

Device implementation:

```
exports.createDevice = function(base) {
  const logger = base.logger || host.logger
  // ...

  function setValue(params) {
    logger.log(params.Value)
  }

  return { setup, start, stop, setValue }
}
```

Caution: `params.Value` is case sensitive! `params.value` would return `undefined`.

4.7.1.2 Specific command enum parameters

The CS host pre-processes the values of enum parameters for a device command function in such way that the device command function always receives a `string` or a `number` depending on the configuration and

regardless how the client application triggered the device commmand.

Example:

- If the driver has the following configuration:

```
 {
// other configurations...
"commands" : [
  {
    "name": "Set Power",
    "params": [
      { "name": "Status", "type": "enum", "enums": ["Off", "On"]}
    ]
  }
],
"variables" : [
  {
    "name": "Power",
    "type": "enum",
    "enums": ["Off", "On"],
    "perform":  { "action": "Set Power", "params" : { "Status":"$string" }}
  }
]
}
```

- And a client app triggers `perform( 'Projector', 'Set Power', { Status: 1 } )`.
- The device command function receives a `params.Status` with a value of `'On'` because Overture CS translated `{ Status: 1}` to `{ Status: 'On'}`.

```
function setPower(params) {
  console.log(params.Status)  // => 'On'
}
```

IMPORTANT:

Special care must be taken when one of the command parameter is of type `enum` and:

- there's no variable associated to the command via the `perform` key of the variable
- or the driver doesn't use device specific command functions but rather the generic device `perform()` function.

In these cases, the actual value of the parameter can be either a `string` and or `number` depending on how the the client application calls the device commands. The driver code must handle both cases.

Example:

If the device command matches the above conditions:

If the application code calls `perform('Projector', 'Set Power', { Status: 'On'})` in a control panel. The device command function receives a `string` with a value of `'On'`.

```
function setPower(params) {
  console.log(params.Status)  // => 'On'
}
```

If the application code calls `perform('Projector', 'Set Power', { Status: 1 })` in a control panel.

The device command function receives a `number` with a value of `'1'`.

```
function setPower(params) {
  console.log(params.Status)  // => 1
}
```

4.7.2 device.perform()

The base device tries to call the `device.perform()` function when it doesn't find a function matching the camel-cased version of the perform action (as discussed above).

In that case the device `perform()` function receives 2 parameters:

- `action`: the action to perform
- `params`: the parameters for the perform command

Example:

```
exports.createDevice = function(base) {

  // ...

  function perform(action, params) {
    if (action === 'Do Something') {
      doSomething(params)
    }
  }

  return { start, stop, tick, perform }
}
```

The `base`device throw an error message if neither a matching specific command nor `perform()` is defined in the driver device.

4.7.3 Command queuing

The `base` device automatically queues external commands from client application and calls the internal driver device command handlers only when the previous command is considered as 'done'.

4.7.3.1 Fire and forget protocols

The command queuing mechanism is transparent for drivers which don't need to check the response from the external device before sending the next command.

In the following example, the 'Set Shutter' command is considered as 'done' as soon as the `setShutter` function returns.

Example:

```
exports.createDevice = function(base) {
  // the socket which will be used by this device
  let socket

  // handles perform('Set Shutter') external requests
  function setShutter(params) {
    if (params.Status === 'Open') {
      socket.send('SHUTTER OPEN\n')
    }
    else if (params.Status === 'Closed') {
      socket.send('SHUTTER CLOSE\n')
    }
  }

  return { /* exported functions */ }
}
```

4.7.3.2 Command/Response protocols

Some protocols require that a driver waits until a response comes back from the real world device before sending the next command. In that case, the driver must use the `commandDefer()` , `commandDone()` and `commandError()` functions provided by the `base` device.

In the following example, the 'Set Shutter' command is considered as 'done' when an acknowledgement is received from the external device.

Example:

```
exports.createDevice = function(base) {

  let socket

  // handles perform('Set Shutter') external requests
  function setShutter(params) {
    // we need to wait for a response, so we defer the command termination
    base.commandDefer()
    if (params.Status === 'Open') {
      socket.send('SHUTTER OPEN\n')
    }
    else if (params.Status === 'Closed') {
      socket.send('SHUTTER CLOSE\n')
    }
  }

  function start() {
    socket = createSocketAndConnect(host,port)
    socket.on('data', (data) => {
      data = data.toString()
      // tells that the command is done when an ack or nack is received from the external devic
e
      if (data === 'ACK') {
        base.commandDone()
      } else if (data === 'NACK') {
        base.commandError()
      }
    })
  }

  return { setShutter, start, /* other exported functions */ }
}
```

4.7.3.3 Command Timeout

The driver can specify a timeout when a command completion is deferred. This is done by specifying a `timeout` parameter (in milliseconds) when calling the `base.commandDefer()` function. If the `timeout` parameter is not specified, the system uses a default timeout value (more than 1 minute).

The system throws an error if the timeout is reached before `base.commandDone()` or `base.commandError()` is called. It then processes the next command in the command queue

Example:

```
exports.createDevice = function(base) {
  // ...

  // handles perform('Set Shutter') external requests
  function setShutter(params) {
    // we need to wait for a response, so we defer the command termination
    // and we set the timeout to 1 second
    base.commandDefer(1000)
    // ...
  }

  //..

  return { setShutter, /* other exported functions */ }
}
```

4.7.3.4 Checking the pending command

A driver can get a reference to the current pending commmand by calling the `base.getPendingCommand()` function. The returned value is falsy if there's no pending commmand.

The most useful properties of the pending command object are:

- `action`: the action of the command
- `params`: the parameters of the command

These properties can be used when working with protocols where responses contain only values and no context. For example, let's say that a protocol accepts 2 different queries but answers only with values:

- request: `?Brightness\n` => response: `80\n`
- request: `?Temperature\n` => response `75\n`

```
// process a frame received from the real world device
function onFrame(frame) {
  const value = Number(frame)
  const pendingCommand = base.getPendingCommand()
  if (pendingCommand) {
    switch (pendingCommand.action) {
      case 'Get Brightness': base.getVar('Brightness').value = value; break;
      case 'Get Temperature': base.getVar('Temperature').value = value; break;
    }
  }
}
```

4.7.4 Synchronous Result

A driver device specific command function can return a synchronous result by using a `return` statement at the end of the function. The return value can be an arbitrary Javascript value (number, string, object, etc.). In that case, this result value is returned to the client application or client object.

The function is considering as returning no value if it either

- doesn't have a return statement
- or returns `undefined`.

4.7.5 Asynchronous Result

In some advanced scenarios, a device specific command can only return a result aynchronously. This happens, for example, when a command can return a result only after it has received a response to a request triggered by the commmand from an external device.

In that case, the command specific function must:

- call the `base.commandDefer()` function in the command handler
- call the `base.commandDone()` function when a result is available.

The following code illustrates this mechanism:

```
exports.createDevice = function(base) {
  // ...

  function setValue(params) {
    // make sure to flag this command as being asynchronous
    base.commandDefer()
    // perform some asynchronous action
    getAsynchronousValue(params)
  }

  function getAsynchronousValue {
    // a real code would issue some kind of external request
    setTimeout( () => {
      // call commandDone() when a result is available
      base.commandDone('An asynchronous value!')
    }, 1000)
  }

  return { setValue, /* other exported functions */ }
}
```

4.7.6 Optimizing Command Queuing

A driver writer can specify whether the command queuing process should be optimized for a specific command by setting the `optimize` key of a command object in the `package.json` file.

This is specially useful when a control panel uses a slider widget to monitor/control a variable of a device which is slow to acknowledge the command. In that case (and if `optimize` is not set), commands tend to accumulate in the command queue when the user moves the slider and the users has the feeling that the system is 'slow'.

The optimize property can be either a boolean value or a string array.

Boolean:

```
{
  "commands" : [
    {
      "name": "Set Level",
      "params": [
        { "name": "Level", "type": "integer" }
      ],
      "optimize": true
    }
    // other commands...
  ]
}
```

String Array:

```
{
  "commands" : [
    {
      "name": "Set Channel Level",
      "params": [
        { "name": "Channel", "type": "integer" },
        { "name": "Level", "type": "integer" }
      ],
      "optimize": ["Channel"]
    }
    // other commands...
  ]
}
```

4.7.6.1 Algorithm for a command optimization:

- if a command has the `optimized` property and there is already the 'same' command in the command queue: the new command replaces the existing queued command.
- a command is said to be the 'same' if:
  - the `optimize` property is a boolean with a value of `true` and the command name is the same
  - the `optimize` property is a string array, the command name is the same and all of the command parameters which are specified in the string array have the same value.

4.7.6.2 Example 1: A slider sets the brightness of a projector

Initial Conditions:

- a slider is linked to the `Brightness` variable of a projector
- The command to set the brightness is `perform('Set Brightness', { Value: brightness_value })`
- This command has `optimize` set to `true`

Execution:

- the user moves the slider: a first command `perform('Set Brightness', { Value: 10 })` is executed and set to 'pending' as long as the projector does not acknowledge it.
- the user keeps on moving the slider: a second command `perform('Set Brightness', { Value: 20 })` is queued but not executed because the first command is still 'pending'
- the user keeps on moving the slider: a third command `perform('Set Brightness', { Value: 30 })` is received and replaces the currently queued command (the second command)

- the user keeps on moving the slider: a fourth command `perform('Set Brightness', { Value: 40 })` is received and replaces the currently queued command (the third command)
- the projector acknowledges the first command
- the command manager executes the next command in the queue which is `perform('Set Brightness', { Value: 40 })` (the fourth command received by the command manager)

4.7.6.3 Example 2: A slider sets the volume of a channel of an audio matrix

Initial Conditions:

- a slider is linked to the `Out1_Volume` variable of an audio matrix.
- The command to set the channel volume is `perform('Set Volume', { Channel: channel_id, Level: channel_level })`
- This command has `optimize` set to `['Channel']`

Execution:

- the user moves the slider: a first command `perform('Set Volume', {Channel: 1, Value: 10 })` is executed and set to 'pending' as long as the projector does not acknowledge it.
- the user keeps on moving the slider: a second command `perform('Set Volume', { Channel: 1, Value: 20 })` is queued but not executed because the first command is still 'pending'
- the user keeps on moving the slider: a third command `perform('Set Volume', { Channel: 1, Value: 30 })` is received and replaces the currently queued command (the second command) because the Channel parameter is the same.
- the user keeps on moving the slider: a fourth command `perform('Set Volume', { Channel: 1, Value: 40 })` is received and replaces the currently queued command (the third command) because the Channel parameter is the same.
- the projector acknowledges the first command
- the command manager executes the next command in the queue which is `perform('Set Volume', { Channel: 1, Value: 40 })` (the fourth command received by the command manager for Channel 1)

Note: another user can still send a 'Set Volume' command with another Channel parameter. This command will be queued independently.

4.7.7 Status Polling

Getting the status of a real world device by regularly sending polling requests is a very common scenario. The base device class embeds a polling manager which eases the implementation of status polling commands (see IBaseDevice Interface). Basically you tell the polling manager what function of your driver should be regularly called and the polling manager takes care of the command scheduling, command queuing and other bookkeeping tasks.

The `base.startPolling()` must be called when the device starts. The polling manager will then call the specific device driver function(s) specified via the `base.setPoll()` call. Note that `base.setPoll(options:IPollingRequestOptions)` is more flexible and allows to poll immediately after a `startPolling()` call. Otherwise, the polling request is executed after a delay (matching with the `IPollingRequestOptions.period`) after the `startPolling()` execution. The `startPolling()` function accepts an `options` parameter of type `IStartPollingOptions`. Currently this `options` setting allows to specify the processPeriod (by default 100ms) which is the period to unstack polling commands waiting for processing.

```
interface IStartPollingOptions {
  processPeriod?: number
}

interface IPollingRequestOptions  {
  action: string,
  period?: number, //in ms (5000ms by default)
  params?: any,
  enablePollFn?: any,
  startImmediately: boolean //false by default
}
```

For example:

```
exports.createDevice = function (base) {
  // tell the polling manager to send a 'Get Status' every 10 secs
  base.setPoll('Get Status', 10000)
  //...

  function getStatus() {
    socket.send('Status?\r\n')
  }

  function start() {
    base.startPolling()
  }

  return { start, getStatus, /* and other exported functions */ }
}
```

You can even pass a `params` object and an `enablePollFn` function to customize the polling manager behavior.

Example:

```
exports.createDevice = function (base) {
  // tell the polling manager to send a 'Get Channel Status' for channel 1 every 10 secs
  // but only if the device is powered on
  base.setPoll('Get Channel Status', 10000, { Channel: 1 }, checkIfPowerOn);
  //...

  function getChannelStatus(params) {
    tcpClient.write('Status' + params.Channel + '?\r\n')
  }

  function checkIfPowerOn() {
    return base.getVar('Power').value === 1
  }

  function start() {
    base.startPolling()
  }

  return { start, getChannelStatus, /* other exported functions */ }
}
```

4.7.8 Tick function

The driver host regularly calls the `tick()` function of a driver (if present and if the device is started) every 5 seconds. The tick period can be modified by calling `base.setTickPeriod()` specifying milliseconds. The `tick()` function can be used by the driver to process any background tasks.

Note that a driver should implement the `tick()` function rather than calling `setInterval()` by its own. This is because the driver host takes care of the various bookeeping chores and calls the `tick()` function only when relevant.

```
exports.createDevice = function (base) {
  // override the default tick period
  base.setTickPeriod(5000)

  function tick() {
    // do something regularly every 5 sec
  }

  return { tick, /* and other exported functions */ }
}
```

4.8 Runtime

4.8.1 device.start()

The device `start()` function is called when:

- OvertureCS starts
- or when the device activity is enabled again after being disabled
- or after OvertureCS has stopped the device because it had to update the device (because of changes in the UX database for example)

It is important that system resources like sockets, memory and files be allocated only in the `start()` function and not when the device object is created. This is due to the fact that a driver can be loaded by UX for configuration purposes and should not be 'active' in that case.

4.8.2 device.stop()

The device `stop()` function is called:

- when the device activity is disabled (i.e: the device `Activity` variable has been set to 'Off' by a client application)
- or when OvertureCS needs to update the device (because of changes in the UX database for example)

It is important that system resources like sockets, memory and files be de-allocated by `stop()` function.

4.9 Using external NPM modules

OvertureCS device drivers can make use of existing NPM modules. There are a few things to keep in mind when using these modules:

- make sure there are no native dependencies otherwise the driver won't load correctly
- make sure to include all the content of the `node_modules` folder in the the driver package zip file. The easiest way is to use `npm install [amodule] --save` to install the external NPM module in the development phase. Then always do a `npm install` before creating (zipping) a driver package.

- try to require the dependency modules in the `start()` function (instead of the putting the `require()` statements at the top of the driver file).

Example of a (hypothetical and simplistic) Phillips Hue driver which allows turning Hue lamps on or off:

```
exports.createDevice = function (base) {

  let hue
  let HueApi
  let lightState

  let config
  let api
  let state

  function setup (_config) {
    config = _config
  }

  function start()  {
    // require the external module as late as possible
    hue = require('node-hue-api')
    HueApi =  hue.HueApi
    lightState = hue.lightState

    api = new HueApi(config.host, config.username)
    state = lightState.create()
  }

  function setLightOnOff (params) {
    if (params.State === 'On') {
      api.setLightState(params.lampId, state.on())
    }
    else if (params.State === 'Off') {
      api.setLightState(params.lampId, state.off())
    }
  }

  return { start, setLightOnOff, /* and other exported functions */ }
}
```

4.10 Using TypeScript

A driver author can use Microsoft TypeScript to write a OvertureCS driver.

The OvertureCS SDK provides a definition `sdk/interfaces.ts` file which includes TypeScript signatures (interfaces) for all of the objects and functions provided by the SDK. The TypeScript definition file can be imported in a TypeScript file using the `import` statement.

4.10.1 TypeScript configuration file

The following `tsconfig.json` file can be used as a basis for the TypeScript configuration file.

```
{
    "compilerOptions": {
        "target": "es2015",
        "module": "commonjs",
        "sourceMap": true,
        "watch": false,
        "allowJs": false
    },
    "exclude": [
        "node_modules"
    ]
}
```

4.10.2 Compiling TypeScript

- cd into the directory of the driver
- type `tsc`

Alternatively, the TypeScript compiler can be run in the background and whatch for file changes.

- cd into the directory of the driver
- `tsc -w`

5 Behaviors

This section applies to OvertureCS version 1.2.0 or greater.

A Behavior is a special kind of driver whose purpose is to implement some domain specific logic. Behaviors can be seen as 'scripts' attached to logical points of an Overture project. A typical behavior would be a script attached to a point of type 'room' and which implement whatever logic is required for this room. But a behavior can be attached to any 'container' point, such as a floor, a building or a custom defined point type.

The Behavior 2.0 functionality allows composing several behaviors and attaching them to a single point (typically a 'room' point). This feature is available since OvertureCS version 1.4.0.

This section cover specific topics related to behaviors.

5.1 Specific package.json configuration

There are very few differences between a behavior package.json and device driver package.json:

- `overture.type` key: must be `behavior` for behaviors
- `overture.brand` key: usually set to `overture` for behaviors
- `overture.models` key: usually set to an empty array for behaviors

5.2 Setting other point variables

Most of the times a behavior interacts with other entities of an Overture project by setting variable values. This can be done using the `host.setVariable(point, value)` function. Note however that only points managed by the same OvertureCS control server can be set.

`setVariable()` accepts 2 parameters:

- `point`: a point reference which specifies which variable(s) must be set. See Point References
- `value`: the value to set

Example:

```
host.setVariable('ControlServer.Malaga_Projector.Power', 'On')
```

Full index.js of a (somewhat contrived) example:

```
{
  const host
  function init(_host) {
    host = _host
  }

  function createDevice(base) {
    function setup() {}
    function start() {}
    function stop() {}
    function setMode(params) {
      const power = params.Value === 'On' ? 'On' : 'Off'
      host.setVariable('Malaga_Projector.Power', power)
    }
    return { start, stop, setMode }
  }
}
```

5.3 Sending Perform to other points

Behaviors can use `host.perform(point, action, params)` to send commands to other points of the same CS.

`perform` accepts 3 parameters:

- `point`: a point reference which specifies which point the command must be sent to. See Point References
- `action`: the action of the command
- `params`: the parameters of the command

Example:

```
host.perform('ControlServer.Malaga_Projector', 'Set Power', { Status: 'On' })
```

Example of full index.js

```
{
  const host
  function init(_host) {
    host = _host
  }

  function createDevice(base) {
    function setup() {}
    function start() {}
    function stop() {}
    function setMode(params) {
      const power = params.Value === 'On' ? 'On' : 'Off'
      host.perform('Malaga_Projector', 'Set Power', { Status: power })
    }
    return { start, stop, setMode }
  }
}
```

5.4 Point References

A point reference can be either:

- an absolute reference using the full path of a point. Ex: `Malaga_Projector.Power`.
- a "point filter" which allows selecting one or more points based on the criteria of a "filter". Ex: `{ parent: ${base.name}, type: 'variable', variablename: 'Power', depth: 2}`

Note that a point reference must refer to points which are managed by the same control server as the one running the behavior. Inter-control-servers references are not supported yet in the current version of OvertureCS.

5.4.1 Point Filter

Note point filter are available starting from OvertureCS version 1.3.0.

A point filter is a set of criteria used to select one or more points. Point filters are useful if the same value or command must be applied to a set of points. In particular, point filters allows selecting point relative to a 'parent' point. This in turn makes possible to write a 'generic' behavior which can be attached to different points.

The various criteria of the filter are expressed using keys of a Javascript object.

Example of a point filter which selects all of the variables which

- have the current point as an ancestor AND
- have the 'av' tags:

```
{
  parent: ${base.name},
  depth: 100,
  tags: ['av']
}
```

5.4.2 Point Filter keys

- `parent`: specifies the ancestor of the points to select
- `depth`: specifies how many levels the search should go while traversing the point hierarchy. If `parent` is specified and `depth`is not specified. The search algorithm assumes a default value of `1` (one level deep). `depth` must be a positive number from `1` to `100`.
- `variablename`: specifies a string which must part of a point's variable name: ex: `.Status`.
- `type`: specifies the type of the point. ex: `'variable'`.
- `subtype`: specifies the sub-type of the point. ex: `'integer'`.
- `tags`: specifies the roles the point must have. ex: `['av', 'lobby']`. Note: the point is selected if it has ALL of the tags of the list.
- `roles`: specifies the roles the point must have. ex: `['tech']`. Note: the point is selected if it has ALL of the roles of the list.

A point is selected if it satisfies ALL of the criteria of the filter.

5.4.3 Advanced Point Filter

The `parent` key can also be an array. In that case, each element of the array describes the parent of the point to be selected.

The following filter selects all of the points which

- have `.Power` in their variable name
- have the 'Floor1' point as an ancestor (that is which are located at floor #1) AND
- are direct children of a projector device

```
{
  variablename: '.Power',
  parent: [
    {
      variablename: 'Floor1',
      depth: 100
    },
    {
      type: 'device',
      subtype: 'projector',
      depth: 1
    }
  ]
}
```

5.5 Monitoring other point variables

Another use case behaviors have to implement is to react to external variable changes. The 'classic' example is doing something when an hardware button is pressed. The `base` object provides 2 functions which allows handling variable changes.

- `base.addVariableListener(variableName, event, callback)`: subcribe to variable changes events.
- `base.removeVariableListener(variableName, event, callback)`: unsubcribe to variable changes.

These functions are formally described in the IBaseDevice Interface but we'll provide a full example in this section:

Let's say a behavior must power a projector on or off when a button is pressed on a hardware control panel. The hardware control panel is monitored by a device driver of type 'I/O' via the 'DigitalInput1' variable of a device named 'IO'. The following code implement this use case.

```
{
  const host
  function init(_host) {
    host = _host
  }

  function createDevice(base) {
    function setup() {}

    function start() {
      // subscribe to 'actualchange' event for the variable 'IO.DigitalInput1'
      base.addVariableListener('IO.DigitalInput1', 'actualchange', buttonChanged )
    }

    function stop() {
      // unsubscribe from the 'actualchange' event for the variable 'IO.DigitalInput1'
      // the following line is optional because it is automatically done when the device is sto
pped.
      base.removeVariableListener('IO.DigitalInput1', 'actualchange', buttonChanged )
    }

    // called when 'IO.DigitalInput1.value' changes: powers a projector off or on according to
the actual button state
    function buttonChanged(variable) {
      host.setVariable('Malaga_Projector.Power', variable.value === 0 ? 'Off' : 'On' )
    }

    return { setup, start, stop }
  }
}
```

6 Driver SDK Helpers

The driver SDk provides several helper objects which ease the implementation of drivers.

6.1 Host

6.1.1 IHost Interface

The `host` object (passed as a parameter to the driver `init()` function) exposes several useful objects or functions that the driver can take advantage of:

- `logger`: A logger object which allows logging message on the OvertureCS application log. see ILogger Interface
- `createFrameParser()`: creates a FrameParser instance. see IFrameParser Interface
- `createNetworkUtilities()`: creates a NetworkUtilities instance. see INetworkUtilities Interface
- `createTcpClient()`: creates a TcpClient instance. see ITcpClient Interface
- `perform()`: allows sending a command to another device which exists on the same OvertureCS. Since OvertureCS version 1.5.1 and Overture UX Server Version 3.3.0, it's possible performing a command on a device which is defined on another Control Server, or even on multiple devices by using a filter.

```
host.perform('OtherDevice', 'Some Command', { Value: 'A parameter'})
host.perform('Showmaster_LE_22070186.MAS_Player1', 'Stop')
```

- `setVariable()`: allows setting the required value of a variable which exists on the same OvertureCS. Since OvertureCS version 1.5.1 and Overture UX Server Version 3.3.0, it's possible setting the value of a variable which is defined on another Control Server, or even multiple variables by using a filter.

```
host.setVariable('OtherDevice.Test', 'A string value')
host.setVariable({parent: base.name, type:'variable', variablename:'Power', depth:2}, 'Off')
```

- `getVariable()`: allows getting a reference to a variable (which exists on the same OvertureCS) by specifying its name

```
let variable = host.getVariable('OtherDevice.Test')
```

- `addVariableListener()`: deprecated, see IBaseDevice Interface for more details
- `removeVariableListener()`: deprecated, see IBaseDevice Interface for more details

`IHost` gives access to several NPM modules which are loaded internally in the Control Server.

- `lodash`: a reference to the Lodash NPM module. Lodash is kind of a swiss army knife for JavaScript programming. It provides many utility functions for array, object and time manipulation. See the Lodash WEB site.
- `request`: a reference to the Request NPM module. Request allows to easily send HTTP(S) requests and is particulary useful for controlling devices via a REST API. Note that `IHost` provides a

'promisified' version of Request.

The HTTP/REST Client section shows an example which makes use of `request`.

Formal interface definition (using TypeScript notation):

```
interface IHost {
  // utitilty objects and classes
  logger : ILogger
  createFrameParser : (logger: ILogger) => IFrameParser
  createTCPClient : (logger: ILogger, options?: ITCPClientOptions) => ITCPClient
  createNetworkUtilities : (logger: ILogger) => INetworkUtilities

  // inter-device control
  perform : (object: string | object, action: string, params: any, options?: IPerformOptions) =
> any
  setVariable : (object: string | object, value: any) => void
  getVariable : (name :string) => IVariable
  addVariableListener : (variableName :string, event :string, callback :any, device?: IBaseDevi
ce) => void
  removeVariableListener : (variableName :string, event :string, callback :any, device?: IBaseD
evice) => void

  // popular NPM modules
  lodash: any,
  request: any,
}
```

6.2 Base Device

6.2.1 IBaseDevice Interface

```
interface IBaseDevice {
  name :string

  // variable management
  getVar(name: string): IVariable
  createVariable(options: IVariableConfig): IVariable
  createStringVariable(name: string): IStringVariable
  createEnumVariable(name: string, enums: string[]): IEnumVariable
  createIntegerVariable(name: string, min?: number, max?: number): IIntegerVariable
  createRealVariable(name: string, min?: number, max?: number, precision?: number): IRealVariab
le
  createTimeVariable(name: string): ITimeVariable
  createDateVariable(name: string): IDateVariable

  // command management (CommandManager)
  perform (action :string, params :any) : Promise<any>
  performIfIdle(action :string, params :any) : Promise<any>
  performInPriority(action :string, params :any) : Promise<any>
  commandDefer(timeout?: number) : void
  commandDone(value) : void
  commandError(error) : void
  getPendingCommand() : Command
  clearPendingCommands(): void
  setCommandManagerOptions(options:ICommandManagerOptions) : void

  // polling management
  setPoll(action: string, period: number, params?: Object, enablePollFn?) : void
  setPoll(options:IPollingRequestOptions) : void
  clearPoll(action: string, params: Object) : void
  startPolling(options?:IStartPollingOptions) : void
  stopPolling() : void

  // notification
  addVariableListener(variableName :string, event :string, callback :any) : void
  removeVariableListener(variableName :string, event :string, callback :any) : void

  // others
  setTickPeriod(ms: number)
  setTimeout(callback: (...args: any[]) => void, ms: number, ...args: any[]): any
  clearTimeout(timeoutId: any): void
  setInterval(callback: (...args: any[]) => void, ms: number, ...args: any[]): any
  clearInterval(intervalId: any): void
}
```

Variable Management Details:

The `createVariable()` function accepts an `options` parameter of type `IVariableConfig`. These options match the various settings which are available in the variable key of a package.json file. See overture.variables. This allows settings all of the attributes of a dynamic variable at creation time.

```
interface IVariableConfig {
  name: string
  type: string
  enums?: string[]
  perform?: {
    action: string
    params: Object
  }
  min?: number
  max?: number
  precision?: number
  unit?: string
  icon?: string
  widget?: string
  readonly?: boolean
  humanName?: string
  persistent?: boolean
  smooth?: boolean|number
}
```

Command Management Details:

- `perform`: queues a command to be excuted next
- `performIfIdle`: queues a command to be excuted next only if there's no pending command
- `performInPriority`: pushes a command at the head of the command queue so that the command is executed before all other queued commands
- `commandDefer`: defers the completion of the specified command (until `commandDone` is called). An optional `timeout` parameter can be specified.
- `commandDone`: marks the current pending commmand as 'done' and processes the next commmand in the queue
- `commandError`: marks the current pending command as 'in error' and processes the next command in the queue
- `getPendingCommand`: returns the current pending command (null if there's no pending command)
- `clearPendingCommands`: clears the command queue
- `setCommandManagerOptions`: defines specific options to the Command Management like `timeBetweenCommands` which allows to set a delay between each commands.

```
interface ICommandManagerOptions {
  timeBetweenCommands?: number,
}
```

Polling Management Details:

- `setPoll`: adds/modifies an automatic poll.
- `clearPoll`: clear an automatic poll.
- `startPolling`: starts the polling process
- `stopPolling`: stops the polling process

Notification Details:

- `addVariableListener()`: allows to be notified when a variable (which exists on the same OvertureCS) changes. The called must specify the name of the variable to monitor, the event and a callback which will be called when the variable emit the specified event. Two events are currently supported:
  - 'actualchange' : The variable's value has changed.
  - 'requiredchange' A new value change has been requested for the variable.

All Variable Listeners are automatically removed when the device is stopped. A 'best practice' is to use `addVariableListener()` in the `start` function of the driver

```
const callback = (variable) => { /*type your code here*/ }
base.addVariableListener('OtherDevice.Test', 'actualchange', callback )
```

- `removeVariableListener()`: allows to remove a callback associated to a variable change. This function requires the same parameter than `addVariableListener()`.

```
base.removeVariableListener('OtherDevice.Test', 'actualchange', callback )
```

Others Details:

- `setTimeout()`: calls a function after a specified number of milliseconds. The callback is executed only if the device is started. The timer is automatically cleared when the device is stopped.
- `clearTimeout()`: clears the timer returned by the `setTimeout()` method.
- `setInterval()`: calls a function at specified intervals (in milliseconds). If the device is started, this method will continue executing the callback until `clearInterval()` is called, or the device is stopped.
- `clearInterval()`: clears the timer returned by the `setInterval()` method.

6.3 Variable

6.3.1 IVariable Interface

```
interface IVariable {
  name :string
  fullName :string
  value :any
  required :any
  enums? :string[]
  min? :number
  max? :number
  precision?: number
  getLastChangedTime(): number
}
```

- `name`: the name of the variable. Example: 'Status'.
- `fullName`: the full name of the variable prefixed with the variable parent name (usually the device name). Example: 'Projector1.Status'
- `value`: the actual value of the variable. That is the value which represents the actual value of a property of the real word device controlled by the driver.
- `required`: the required value of the variable.
- `min`: (optional) the minimum value of the variable (only defined for variables of type `integer` and `real`)
- `max`: (optional) the maximum value of the variable (only defined for variables of type `integer` and `real`)
- `precision`: (optional) the precision of real variables as digits after the decimal point. Ex: `precision: 4` => `0.0000` (defaults to 2)
- `enums`: (optional) the list of string values for a variable of type `enum`
- `getLastChangedTime()`: returns a number matching with the date of the last value change. If the variable's value never changed, it returns undefined. (available since OvertureCS version 1.6.0)

6.4 Logger

6.4.1 ILogger Interface

This interface allows logging messages on the OvertureCS log which is displayed in the user interface. Several pre-configured log levels are available:

- `error`
- `alarm`
- `warn`
- `info`
- `debug`
- `silly`

The log level of OvertureCS can be set via the `LOGLEVEL` OvertureCS configuration parameter (environment variables or command line parameter). It concerns only the logs in the console which totally ignores the Log Settings coming from the Overture CS GUI.

All Logs Settings accessible in the Overture CS GUI are only related to what is displayed in the Logs view.

Starting from OvertureCS 1.3.0, each device can have its own logger. The OvertureCS GUI allows to automatically add the name of the device to the message and to enable/disable logs on a per device basis which is really helpful when troubleshooting a large installation. Refer to the example below to see how to take advantage of this feature while still being compatible with versions less than 1.3.0.

6.4.1.1 Signature

```
interface ILogger {
  error(message:string) : void
  alarm(message:string) : void
  warn(message:string) : void
  info(message:string) : void
  debug(message:string) : void
  silly(message:string) : void
}
```

6.4.1.2 Usage Examples

If using 'closure style':

```
exports.createDevice = function(base) {
  /*
   * Get a ref to logger used by the base device (or fallback to the host logger
   * if we're running on OvertureCS version < 1.3.0) and keep the ref in the
   * createDevice function scope
   */
  const logger = base.logger || host.logger

  const anotherFunction = function () {
    // ...

    // log a simple message
    logger.info('This is an info message')
  }
}
```

If using 'class style':

```
exports.createDevice = function(base) {
  return new Device
}

class Device {
  constructor(base) {
    this.logger = base.logger || host.logger
  }

  anotherFunction() {
    // ...

    // log a simple message
    this.logger.info('This is an info message')
  }
}
```

6.5 FrameParser

FrameParser is a utility class which splits incoming data into individual frames. FrameParser can be be configured to use either string separators or regular expressions in order to split a stream of data in frames.

The FrameParser:

- manages partial frame buffering
- manages mutliple frames received in the same chunk
- avoids buffer overflow and emit an `'error'` event it detects such a condition.

Driver code must init the FrameParser and then listen to `'data'` event. `'data'` events are emitted each time a full frame of data is available. Data are pushed in to the parser via the `push()`.

Note: https://regex101.com is a useful site for testing regular expressions.

6.5.1 IFrameParser Interface

```
interface IFrameParser {
  setSeparator(separator :string|RegExp) : void
  setMaxBufferLength(length :number) : void
  on(event: string) : void
  push(data) : void
}
```

6.5.2 Example Code

```
// get a reference to the host in the init function of the driver
let host
exports.init = function (_host) {
  host = _host
}

exports.createDevice = function (base)
  let tcpClient

  // create and init a frame parser
  let frameParser = host.createFrameParser()

  // use a regular expression: frames starts with a \xFF and ends with \xFE
  frameParser.setSeparator(/\xFF.+\xFE/)

  // listen for 'data' events from the frameParser
  // which are triggered each time full frame is received
  frameParser.on('data', (frame) => { parseFrame(frame) })

  function start() {
    if (!tcpClient) {
    tcpClient = host.createTcpClient()

    // push data into the frame parser each time we receive data from TCP
    tcpClient.on('data', (data) => {
      frameParser.push(data)
    })
      // etc.
    }
  }

  function parseFrame(frame) {
    // process the frame
  }

  return { start, /* and other exported functions */ }
}
```

6.6 TCP Client

TCPClient is a class which implements a TCP client with automatic reconnection.

6.6.1 ITCPClient Interface

```
interface ITCPClient extends EventEmitter {
  connect(port: number, host?: string, connectListener? ) : void
  on(event:string, handler:any) : void
  write(data) : boolean
  end() : void
  address() : any
  isConnected() : boolean
  setOptions(options: ITCPClientOptions)
}

interface ITCPClientOptions {
  autoReconnectionAttemptDelay?:number,
  receiveTimeout?:number,
  disconnectOnReceiveTimeout?:Boolean,
  noDelay?:Boolean,
  keepAlive?: Boolean,
  keepAliveInitialDelay?: number
 }
```

Notes:

- `autoReconnectionAttemptDelay` (2000ms by default): This is the delay between a disconnection and a reconnection attempt. If its value is equals or lower to 0, the auto-reconnection is disabled. The auto-reconnection is active only when the device is started.
- `receiveTimeout` (60000ms by default): A receiveTimeout event is emitted by the TCPClient when nothing has been received during the specified period. It allows to detect if ethernet cable has been removed or if the network is not working anymore.
- `disconnectOnReceiveTimeout` (true by default) When a receiveTimeout event is emitted, it disconnects the TCPClient. Then if the auto-reconnection is enabled, the TCPClient will attempt automatically to reconnect.
- `noDelay` (true by default): By default TCP connections use the Nagle algorithm, they buffer data before sending it off. Setting true for noDelay will immediately fire off data each time socket.write() is called.
- `keepAlive` (false by default): Enable/disable keep-alive functionality.
- `keepAliveInitialDelay` (0ms by default): Set keepAliveInitialDelay (in milliseconds) to set the delay between the last data packet received and the first keepalive probe.

6.6.2 Usage Example

A full example is given in the Minimal TCP Client Example section.

Here is skeleton of a TCP Client management.

```
let host
exports.init = function (_host) {
  host = _host
}

exports.createDevice = function (base)
  let config
  let tcpClient

  function setup(_config) {
    config = config
  }

  function start() {
    if (!tcpClient) {
      // create a tcpClient when the device is started for the first time
      tcpClient = host.createTCPClient()

      // data event triggered when data is available on the TCP socket
      tcpClient.on('data', (data) => {
        // process the data coming from the TCP socket
      })

      // connection event triggered when the socket is connected
      tcpClient.on('connect', () => {
        // handle connection event
      })

      // close event triggered when the socket is closed (either on error or intentionnaly)
      tcpClient.on('close', (hadErrors) => {
        // handle close event
      })

      // error event triggered when an error occurs on the socket
      tcpClient.on('error', (error) => {
        // handle error
      })

      // connect to the server specified in the config
      tcpClient.connect(config.port, config.host)

    }
  }

  function stop() {
    // stop the client
    if (tcpClient) {
      tcpClient.end()
    }
  }

  return { setup, start, stop, /* and other exported functions */ }
}
```

6.7 HTTP/REST Client

The Driver SDK host provides a `request` object which allows sending HTTP/REST request to an external device. This object is a promise-based wrapper around the `request` NPM module. For more advanced information, please refer to the Request Github Page.

6.7.1 Simple GET request

```
// driver init function
let request;
exports.init =  function (host) {
    request = host.request;
}

exports.createDevice = function (base) {

  function sendRequest() {
    // send a HTTP GET request and process the response
    request('http://www.google.com')
      .then( function (response, body) {
        if (response.statusCode == 200) {
          process(body);  // process the HTML for the Google homepage.
        }
      })
      .catch( function(err) {
        // process errors here
      })
    }

  function processBody(body) {
    // do something useful
  }

  return { sendRequest, /* and other exported functions */ }
}
```

6.7.2 GET request using Digest Authentication

```
// driver init function
let request;
exports.init = function (host) {
    request = host.request;
}

exports.createDevice = function (base) {

  function sendStatusRequest() {
    const options = {
      method: 'GET',
      uri: config.URL + '/status',
      auth: {
        user: config.user,
        pass: config.password,
        sendImmediately: false
      },
      rejectUnauthorized: false,
      json: true,
      timeout: 10000
    };
    request(options)
      .then(function (response) {
        // process response here
      })
      .catch(function (err) {
        // process errors here
      })
  }

  return { sendStatusRequest, /* and other exported functions */ }
}
```

6.8 NetworkUtilities

NetworkUtilities is an utility class providing different functions for network management and system information retrieval.

The NetworkUtilities can:

- switch on a computer by the LAN (Wake On Lan).
- return the MAC address corresponding to the given IP Address.
- send a ping request to a computer.

6.8.1 INetworkUtilities Interface

```
interface INetworkUtilities {
  wakeOnLan(wolOptions: IWakeOnLanOptions)
  getMacAddress(ipAddress: string)
  ping(host: string, options?: IPingOptions)
}

interface IWakeOnLanOptions {
  ipAddress?: string, //(optional ip address; default is 255.255.255.255)
  mac?: string,
  port?: number       //(optional port; default is 9)
}

interface IPingOptions {
  timeout?: number,   //(default: 4000ms)
  retryCount?: number //(default: 1)
}
```

Notes:

- `wakeOnLan()` function only works when Windows version of OvertureCS is used.
- To use the wakeOnLan function, the Mac Address (mac) or the IP address (ipAddress) must be provided. If both are defined, only the Mac Address is used.
- The `timeout` ping option is set in milliseconds even if its precision is in seconds. Ex: 1250 or 1000 are delivering the same result.

6.9 Checksum

A class which implements most common checksums algorithms

Not implemented yet.

6.10 UDP Socket

A class which implements a UDP sender/receiver.

Not implemented yet.

7 Debugging

Drivers can be debugged on Windows by running the compiled version of Overture Control Server, or on Mac OS and Linux,using the Docker version of Overture Control Server, within Visual Studio Code.

7.1 Configuring the UX Server

- create a new Control Server for test in the Overture Configurator
- give the control server an ID (i.e `Test`)
- attach a Device type point of the UX server project to this Control Server
- in the Server text field, choose Overture Control Server you've just created from the drop down menu
- configure the point so it uses the driver you want to debug

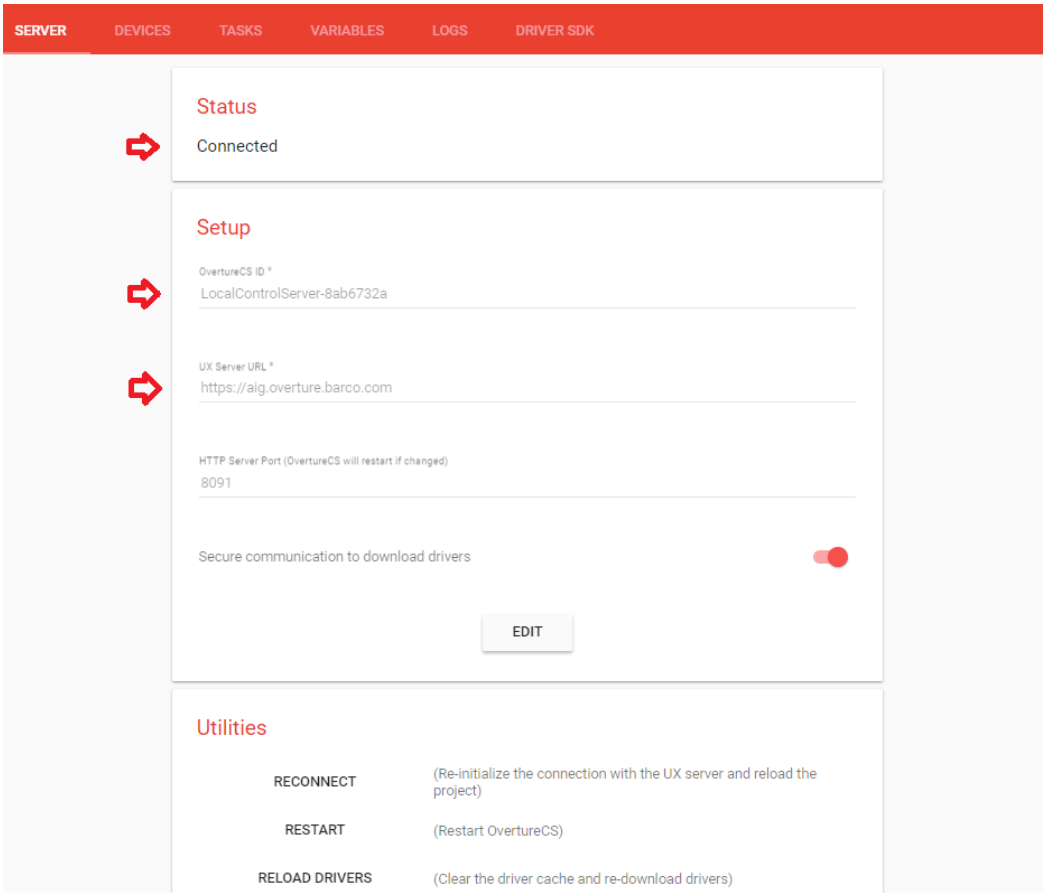
7.2 Debugging under Windows OS

7.2.1 Install OvertureCS

The Windows compiled version of Control Server is available for download at Overture Downloads (Barco account required)

7.2.2 Checking ControlServer operations

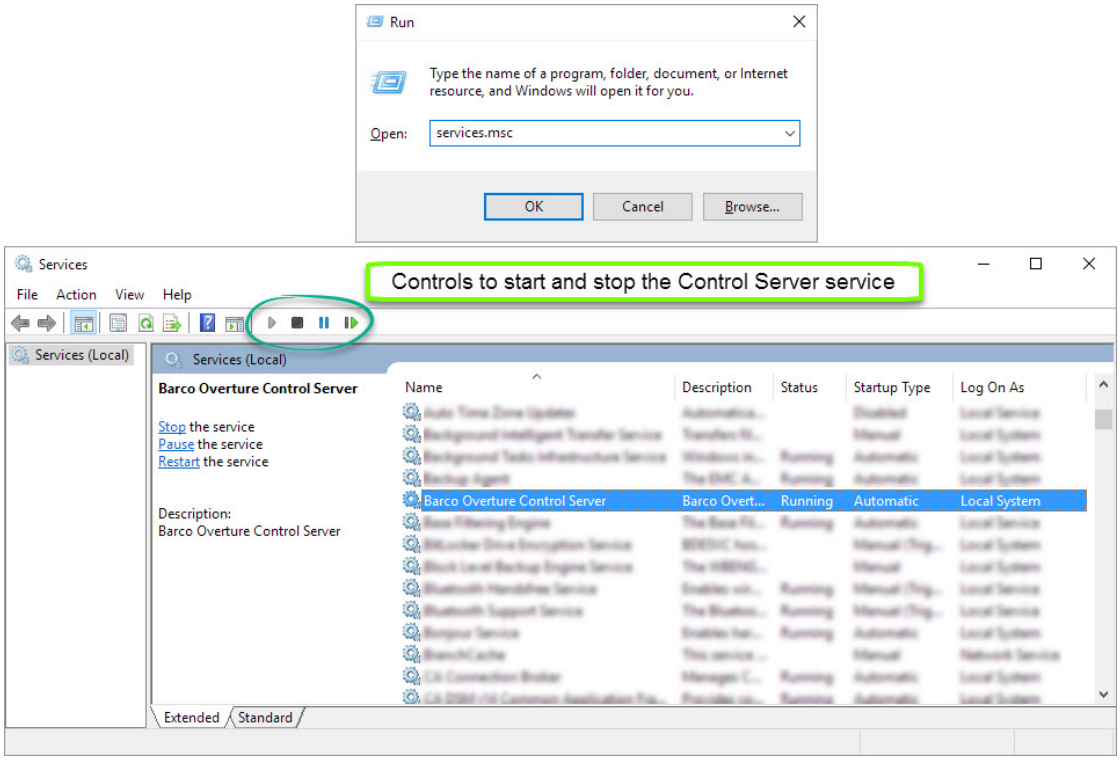
- using the Control Server GUI
- set the Control Server ID (i.e `Test`)
- set the UXServer URL
- Control Server should connect to the UXServer (the GUI should show 'Connected')



- check that ControlServer has created a `drivers` folder at the same level as the application

7.2.3 Disable the Control Server Windows Service

The 'Barco Overture Control Server' service must be stopped before Visual Studio Code debugging can be started.

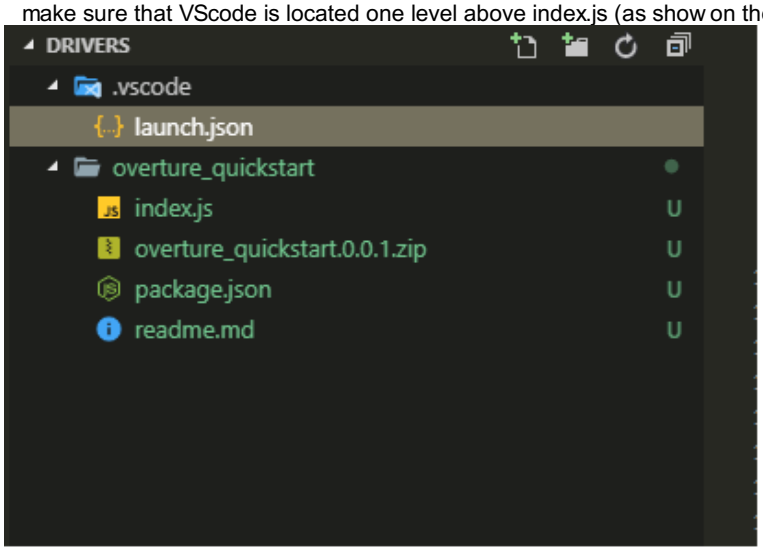


7.2.4 Using Visual Studio Code Debugger

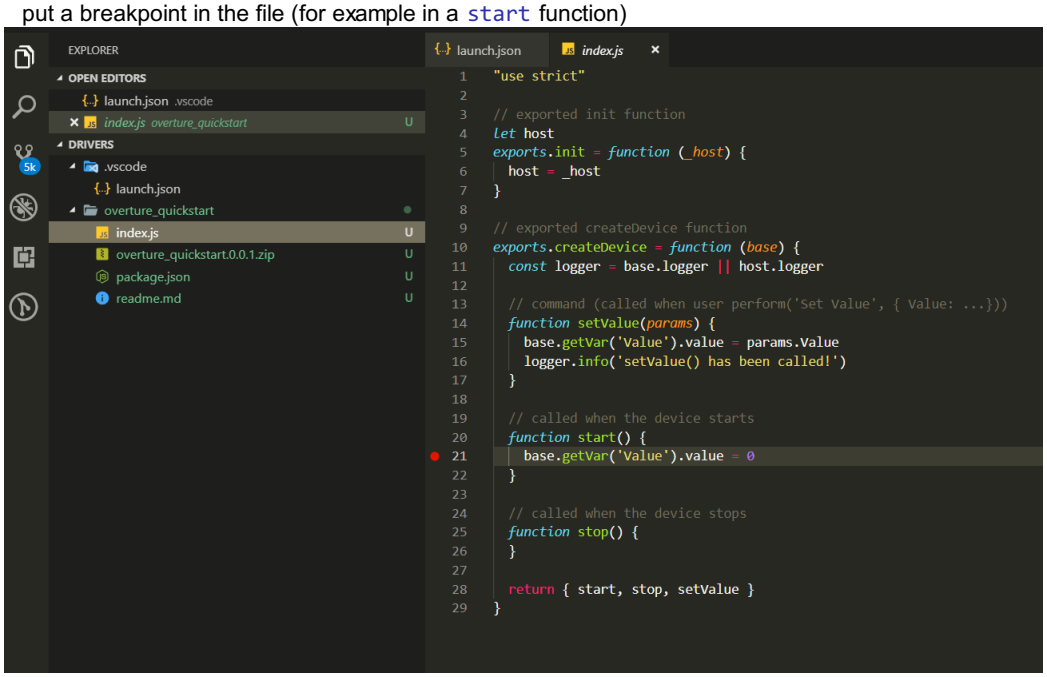
- start Visual Studio Code (depending on your configuration, you may need to "Run as administrator")
- open the the directory where the drivers to debug are installed using the `File/Open...` menu. Ex: "D:\Barco\drivers"
- if not already done, copy and paste the following content in the `.vscode/launch.json` file:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "OvertureCS",
      "runtimeExecutable": "C:\\Program Files\\Barco\\Barco Overture Control Server\\OvertureCS.exe",
      "windows": {
        "runtimeExecutable": "C:\\Program Files\\Barco\\Barco Overture Control Server\\OvertureCS.exe"
      },
      "protocol": "inspector",
      "runtimeArgs": [
        "--inspect=5858"
      ],
      "env": {
        "data": "C:\\Program Files\\Barco\\Barco Overture Control Server\\data\\",
        "driverPath": "${workspaceRoot}",
        "debugDriver": "true",
        "logLevel": "debug"
      }
    }
  ]
}
```

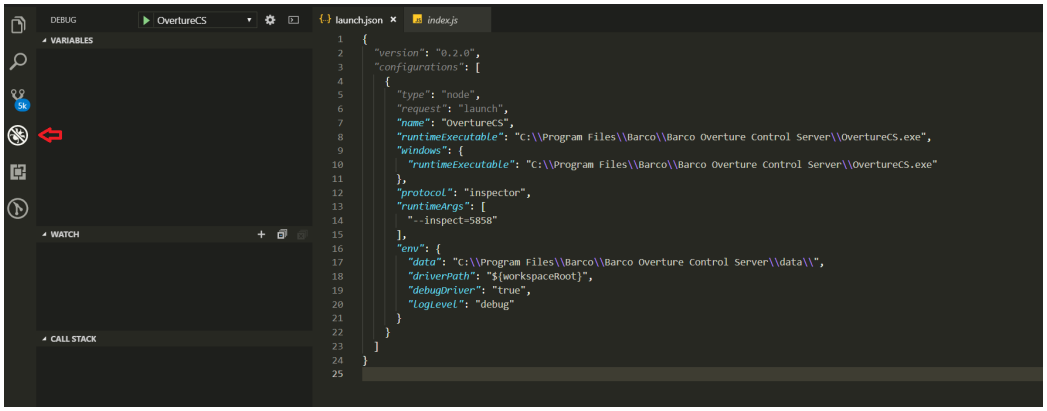
- make sure that VScode is located one level above index.js (as show on the picture down below)



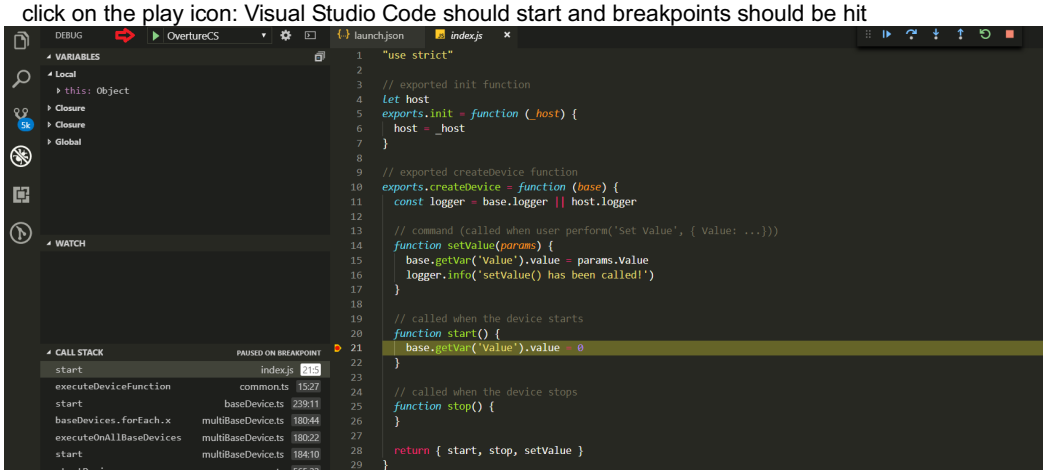
- open the file you want to debug in the Editor
- put a breakpoint in the file (for example in a `start` function)



- switch to the debug view using the `View/Debug` menu
- select `OvertureCS` in the debug configuration select box



- click on the play icon: Visual Studio Code should start and breakpoints should be hit



7.3 Debugging under Mac OS or Linux

7.3.1 Prerequisites

- Docker 17.10.0 or greater installed on your computer

7.3.2 Running Control Server in a Docker Container

7.3.2.1 Downloading and installing the Overture CS Docker image

The Docker version of Control Server is available for download at Overture Downloads (Barco account required)

- download the Docker image file
- open the zip file
- open a command line
- cd in the download directory
- cd in `R330902_06_ApplicationSw`
- type `docker load --input Barco-overtureCS_docker-image-1.3.1.tar`
- type `docker tag barcooverture/controlserver:1.3.1 barcooverture/controlserver:latest`

7.3.2.2 Running the Overture CS Docker image

- create a directory (ie `debug-driver-docker`)
- open this directory with VS Code
- create a file named `docker-compose.yml`
- copy the following content
- replace the `UXURI` key with the URI of your UX Server
- save

```
version: '2'

services:

  # test control server
  cs:
    image: barcooverture/controlserver
    environment:
      - UID=Test
      - UXURI=http://192.168.43.8 # replace with the URI of your UXServer
    volumes:
      - csdata:/data
      - ./:/apps/drivers
    ports:
      - 8080:8080
      - 5858:5858
    command: node --debug index
    restart: always

volumes:
  csdata:
```

- open a command line in the same directory
- type `docker-compose up`
- the Control Server is now running in a Docker container

7.3.3 Debugging with VS Code

7.3.3.1 Starting VS Code

When VS Code is opened, click File -> Open and select the folder created above (ie `debug-driver-docker`) or drag this folder into VS Code to open it.

7.3.3.2 Creating a "Debug Driver in Docker" profile

- switch VS Code to debug view
- click on the gear icon at the top of the screen: VSCode shows the content of launch.json
- replace the existing content with the following content:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug Driver in Docker",
            "type": "node",
            "request": "attach",
            "port": 5858,
            "address": "localhost", // replace with the address of your VM if Docker Toolbox
            "restart": true,
            "sourceMaps": true,
            "outFiles": [],
            "localRoot": "${workspaceRoot}",
            "remoteRoot": "/apps/drivers"
        }
    ]
}
```

- Note: you must change the `address` key to the address of your VM if your running Docker Toolbox instead of Docker Native.

7.3.3.3 Opening the driver source file

- switch to the 'Explorer' view
- now that the Control Server is running, you should see new files in the file pane of VSCode: these are the files of the driver.
- expand the folder of your driver (this driver folder is located in your main folder, ie `debug-driver-docker`)
- open the `index.js` file in that folder

7.3.3.4 Debugging the driver

- switch to the 'Debug' view
- add a breakpoint somewhere in the code (the `tick()` function is a good candidate)
- click on the little green arrow next to the 'Debug Driver in Docker' profile at the top left of the screen
- the VS Code debugger attaches to the Control Server
- the breakpoint is hit
- you can now have the full debugging experience

7.3.4 Notes

- you can make modifications to the source files using VS Code and restart the Docker container: your changes won't be over-written as long as the version in the `package.json` file of the driver is the same as the one of the driver installed in the UX Server. This allows a relatively fast write/test cycle.

7.4 Visual Studio Code Javascript Linting

You may want enhance your development experience and prevent simple syntax errors and typos by using a Javascript Linter. For this purpose we recommend installing ESLint inside VSCode, it is sufficiently easy to install and is well integrated within the IDE. Here are the steps required to have a working linting environment within VSCode:

- Install the eslint executable globaly using npm on a CLI

```
npm install -g eslint
```

- Install the ESLint VSCode extension from the Marketplace (within VSCode, View -> Extensions).
- Enable the installed extension using the Marketplace provided "Enable" button.
- Restart VSCode.
- Create a eslint config file `.eslintrc.json` at the root level of each VSCode Workspace you want to make lintable.

```
{
 "extends": "eslint:recommended",
 "env": {
   "es6": true,
   "node": true
 }
}
```

- ESLint should now show its outputs in VSCode 'problems' console (View -> Problems) as soon as it finds something wrong in your currently opened JS source files.

8 Publishing a driver

When publishing a driver some conventions must be followed in order to make it generally installable on the control server:

- It must be packaged as a Zip archive
- It should contain a package.json file, providing an updated `version` key, incremented according to the Semver version naming scheme, like:
  - `1.0.1`
  - `1.0.2-beta.1`
- The published file archive must as well follow the same version string convention:
  - `<driver_name>.1.0.1.zip`
- The release notes must be updated to reflect your published version change in `readme.md`.
- When providing a custom Control Panel, place its template as well as any required external assets inside a `controlpanel` directory. The provided template name must match the driver name. You can refer to the Overture User Manual to gather further documentation about Control Panels usages.
- The package content organisation would look something like:

```
<driver.name><driver.version>.zip
├── controlpanel (optional)
│   ├── <asset>.png
│   └── <driver.name>.html
├── index.js
├── license.txt (optional)
├── node_modules (optional)
├── package.json
└── readme.md
```

If you use a version control system:

- Commit your changes with the new archive created.
- Tag this commit with the version number of the driver.
- Push your commits with the tags.

8.1 QA Check Points

To ensure the quality of the driver before publishing, there are some important points related to the installation and functionality of drivers that must be validated:

- Driver must get uploaded to Plug-Ins view in Configurator
- Device must get created using the driver
- Device should have a setup page if needed, the setup must be functional
- Device must get loaded on CX unless the CX version is not supported, if not, it must log an error
- The control panel template(distributed or present in driver) must give a useful interface and be functional
- Device should perform all described commands
- Variables must get updated on the device when command is performed on the control panel as well as on the control panel when action is done on the physical device or by third party
- Device must work as expected using the driver
- Device must appear disconnected if the driver cannot communicate with the device anymore and must get connected when communication is reestablished
- Device should appear disconnected if the cable is disconnected from the device and should get connected when the cable is connected back

8.2 Driver Quality Checklist

Driver Quality Checklist is a complete checklist template for driver developers to help ensure the quality of drivers being published. The checklist includes test cases that focus on Creating the driver, Packaging the driver files, Installing the driver in Overture UX, Creating a Device, Loading the device on Overture Control Server and Using the device.
All test cases specified in the Driver Quality Checklist must be tested and all tests must pass before the driver is published. Driver Quality Checklist is the most recent template that should be used for the QA process.

If you want to publish your driver to the online library, send it to the Barco Overture at service.controlrooms.usa@barco.com if you are located in North America or at service.overture.emea@barco.com if you are located elsewhere. Note: an OvertureCS driver dedicated WEB site will allow to upload and manage drivers in the future.

9 Examples

9.1 Quickstart Example

This quickstart example implements a minimal driver which manages only one integer variable (named 'Value') and one command (named 'Set Value').

Both ES2015 and ES5 versions are shown.

It's made of 4 files:

- package.json
- index.js
- readme.md
- license.txt

9.1.1 Quickstart Example: package.json

```
{
  "name": "overture_quickstart",
  "version": "0.0.1",
  "description": "A quickstart Overture device driver",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Barco",
  "license": "SEE LICENSE IN license.txt",
  "overture": {
    "brand": "Overture",
    "models": ["quickstart"],
    "type": "device",
    "subtype": "projector",
    "variables": [
      {
        "name": "Value",
        "type": "integer"
      }
    ],
    "commands": [
      {
        "action": "Set Value",
        "params": {
          "Value": {
            "type:": "integer"
          }
        }
      }
    ]
  }
}
```

9.1.2 Quickstart Example: index.js

```
"use strict"

// exported init function
let host
exports.init = function (_host) {
  host = _host
}

// exported createDevice function
exports.createDevice = function (base) {
  const logger = base.logger || host.logger

  // command (called when user perform('Set Value', { Value: ...}))
  function setValue(params) {
    base.getVar('Value').value = params.Value
    logger.info('setValue() has been called!')
  }

  // called when the device starts
  function start() {
    base.getVar('Value').value = 0
  }

  // called when the device stops
  function stop() {
  }

  return { start, stop, setValue }
}
```

9.1.3 Quickstart Example: readme.md

```
# Quickstart

Driver for a quickstart device.

## Overview

This device has one variable named 'Value' that can be changed by the `Set Value` command.

## Setup

No setup.

## Variables

### Value

The one and only variable of this device.

Type: `integer`

## Commands

### Set Value

## Revisions

### 0.0.1

- initial version
```

9.1.4 Quickstart Example: license.txt

```
© 2017, Barco NV.

The software included in the Device Drivers ("this Software") may be used by you if you have ob
tained a valid license to use Barco Overture and subject to the terms of such Barco Overture Li
cense Agreement.

You are permitted to modify this Software, in both source and object code format, strictly as r
equired for your use of Barco Overture, and without removing or obscuring the above copyright n
otice.

No rights are granted to use this Software, whether modified or as is, outside Barco Overture o
r otherwise than in connection with the Barco Overture product.  All other rights, whether to c
opy, distribute, modify or otherwise, are reserved by Barco.

Barco does not assume any liability in respect of any modifications to this Software.
```

9.2 Minimal TCP Client Example

This example shows a minimal implementation of a simple TCP Client driver. The driver supports one command (`Set Power`) and two status variables (`Status` and `Power`).

The protocol to implement is a simplistic text protocol:

Requests:

- `!Power ON\n`: Powers the device on
- `!Power OFF\n`: Powers the device off
- `?Power\n`: queries the Power status of the device

Responses:

- `?Power ON\n`: returned when device in on in response to the `?Power\n` request
- `?Power OFF\n`: returned when device in off in response to the `?Power\n` request

Note that this example doesn't handle errors, data buffering and is not production ready.

9.2.1 Minimal TCP Client Example: package.json

```
{
  "name": "overture_tcpclient_template",
  "version": "1.0.0",
  "description": "A minimal TCP client device driver template",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Barco",
  "license": "SEE LICENSE IN license.txt",
  "overture": {
    "brand": "Overture",
    "models": ["tcpclient"],
    "type": "device",
    "subtype": "projector",
    "variables": [
      {
        "name": "Status",
        "type": "enum",
        "enums": ["Disconnected","Connected"]
      },
      {
        "name": "Power",
        "type": "enum",
        "enums": ["Off","On"],
        "perform": {
          "action": "Set Power",
          "params": {
            "Status": "$string"
          }
        }
      }
    ],
    "commands": [
      {
        "action": "Set Power",
        "params": {
          "Status": {
            "type": "enum",
            "enums": ["Off","On"]
          }
        }
      }
    ],
    "pointSetupSchema": {
      "type": "object",
      "title": "Driver",
      "properties": {
        "host": {
          "title": "host name or IP address:",
          "type": "string",
          "default": "dev.overture.barco.com"
        },
        "port": {
          "title": "Port:",
          "type": "integer",
          "default": 5888
        }
      },
      "required": [
        "host",
        "port"
      ]
    }
  }
}
```

9.2.2 Minimal TCP Client Example: index.js

```
"use strict"

// exported init function
let host
exports.init = function (_host) {
  host = _host
}

// exported createDevice function
exports.createDevice = function (base) {
  const logger = base.logger || host.logger
  let config
  let tcpClient

  function setup(_config) {
    config = _config
  }

  // connect to the device when start()
  function start() {
    initTcpClient()
    tcpClient.connect(config.port, config.host)
  }

  // disconnect from the device when stop()
  function stop() {
    tcpClient.end()
  }

  // regularly poll the power status of the device
  function tick() {
    tcpClient.write('?Power\n')
  }

  // send a 'power' command
  function setPower(params) {
    if (params.Status === "On") {
      tcpClient.write('!Power ON\n')
    } else if (params.Status === "Off") {
      tcpClient.write('!Power OFF\n')
    }
  }

  // create a tcp client and hook event handlers
  function initTcpClient() {
    if (!tcpClient) {
      tcpClient = host.createTCPClient()
      tcpClient.on('data', onData)
      tcpClient.on('connect', onConnect)
      tcpClient.on('close', onClose)
    }
  }

  function onData(data) {
    data = data.toString()
    // update the power status depending on the data
    if (data === '?Power ON\n') {
      base.getVar('Power').value = 1
    } else if (data === '?Power OFF\n') {
      base.getVar('Power').value = 0
    }
  }

  function onConnect() {
    base.getVar('Status').value = 1
```

```
  }

  function onClose() {
    base.getVar('Status').value = 0
  }

  // publish public functions
  return { setup, start, stop, tick, setPower }
}
```

9.2.3 Minimal TCP Client Example: readme.md

```
# TCP Client Example


## Overview

Showcases a minimal implementation of a driver which control a real world device via TCP.

## Setup

- `port`:  (number) the port of the device
- `host`:  (string) the hostname or the IP address of the device

## Variables

- `Status`:

### Status

The connection status of the device.

Type: `enum`

Enums:

- "Disconnected" (0) The device is not connected
- "Connected" (1) The device is connected

### Power

The power status of the device

Type: `enum`

Enums:

- "Off" (0) The device is powered off
- "On" (1) The device is powered on

## Commands

### Set Power

Parameters:

- `Status`:
  - type: `enum`
  - enums:
    - `Off`: turns the device power on
    - `On`

## Revisions

### 0.0.1

- initial version
```

9.3 Dynamic Variables Example

For some drivers, the number of device variables depends upon the setup of a device. A typical example is a lighting system which can manages a variable number of lights.

9.3.1 Dynamic Variables Example: package.json

```
{
  "name": "dynamic_variables_driver",
  "version": "0.0.1",
  "description": "A driver which creates dynamic variables",
  "scripts": {
    "start": "node index.js"
  },
  "main": "index.js",
  "author": "Barco",
  "license": "SEE LICENSE IN license.txt",
  "overture": {
    "brand": "Overture",
    "models": ["Example"],
    "type": "device",
    "subtype": "lighting",
    "variables": [],
    "commands": [
      {
        "action": "Set Level",
        "params": {
          "Channel": { "type": "integer" },
          "Value": { "type": "integer" }
        }
      }
    ],
    "pointSetupSchema": {
      "type": "object",
      "title": "Driver",
      "properties": {
        "channelCount" : {
          "title": "Channel Count:",
          "type": "integer",
          "default": 4
        }
      },
      "required": ["channelCount"]
    }
  }
}
```

9.3.2 Dynamic Variables Example: index.js

The most important bit is the `setup()` function which creates variables depending on the setup configuration.

```
"use strict"

// exported init function
let host
exports.init = function (_host) {
  host = _host
}

// driver device implementation
exports.createDevice = function(base) {
  const logger = base.logger || host.logger
  let socket

  function setup(config) {
    // create dynamic variables  depending on the setup configuration
    // this will create the "LevelChannel1", "LevelChannel2", etc. variables
    // and link them to the the "Set Level" command
    for( let i = 0; i < config.channelCount; i++) {
      let channel = base.createIntegerVariable('LevelChannel' + (i+1), 0, 100)
      channel.perform = { action: 'Set Level', params: {Channel: i + 1, Level: "$value"}}
    }
  }

  // set the level of one channel
  function setLevel(params){
    // send the command to the real world device
    socket.write('SET LEVEL ' + params.Channel + ' ' + params.Level + '\r\n')
  }

  function start() {
    // to implement
  }

  function stop() {
    // to implement
  }

  return { setup, start, stop, setLevel, /* and other exported functions */ }
}
```

10 Tips and Tricks

10.1 Updating the strings of an Enum variable

Sometimes a driver needs to update the string values of an enum variable depending on information obtained from the controlled device. This can be achieved by merely affecting a new string array to the `enums`.

```
// compute the new enum strings
let newPlaylist = getPlaylistFromDevice()
// then update the enums of the variable
base.getVar('Playlist').enums = newPlaylist
```

Note that doing a direct `push()` on the variable enums also updates the variable but doesn't send a change notification to Overture. The affectation operator must be used instead if a notification is needed (which is usually the case) as in the following example:

```
// compute the new enum strings
let newMedia = getNewMediaFromDevice()
let playlist = base.getVar('Playlist').enums
playlist.push(newMedia)
// then update the enums of the variable
base.getVar('Playlist').enums = playlist
```

10.2 Using ES5

Although the code examples of this document are written in ES2015, nothing prevent a driver author to stick to ES5.

The usual changes are:

- replace `let` or `const` by `var`
- replace arrow functions by regular functions
- dont use ES2015 shorthands when returning the published function object

Below is the simple TCP Client example slightly modified in order to use only the ES5 syntax.

```
"use strict"

// exported init function
var host
exports.init = function (_host) {
  host = _host
}

// exported createDevice function
exports.createDevice = function (base) {
  var logger = base.logger || host.logger
  var config
  var tcpClient

  function setup(_config) {
    config = _config
```

```
  }

  // connect to the device when start()
  function start() {
    initTcpClient()
    tcpClient.connect(config.port, config.host)
  }

  // disconnect from the device when stop()
  function stop() {
    tcpClient.end()
  }

  // regularly poll the power status of the device
  function tick() {
    tcpClient.write('?Power\n')
  }

  // send a 'power' command
  function setPower(params) {
    if (params.Status === "On") {
      tcpClient.write('!Power ON\n')
    } else if (params.Status === "Off") {
      tcpClient.write('!Power OFF\n')
    }
  }

  // create a tcp client and hook event handlers
  function initTcpClient() {
    if (!tcpClient) {
      tcpClient = host.createTCPClient()
      tcpClient.on('data', onData)
      tcpClient.on('connect', onConnect)
      tcpClient.on('close', onClose)
    }
  }

  function onData(data) {
    data = data.toString()
    // update the power status depending on the data
    if (data === '?Power ON\n') {
      base.getVar('Power').value = 1
    } else if (data === '?Power OFF\n') {
      base.getVar('Power').value = 0
    }
  }

  function onConnect() {
    base.getVar('Status').value = 1
  }

  function onClose() {
    base.getVar('Status').value = 0
  }

  // publish public functions
  return {
    setup: setup,
    start: start,
    stop: stop,
    tick: tick,
    setPower: setPower
  }
}
```

10.3 Common Pitfalls and Errors

10.3.1 Forgetting to publish public functions

It is very easy to forget to publish public functions in the object returned by `createDevice()`. One rule of thumb is to always check the returned object properties when adding new commands.

Example: BEFORE adding a new driver command

```
"use strict"

// exported init function
let host
exports.init = function (_host) {
  host = _host
}

// exported createDevice function
exports.createDevice = function (base) {
  const logger = base.logger || host.logger
  let config
  let tcpClient

  function setup(_config) {
    config = _config
  }

  // connect to the device when start()
  function start() {
    initTcpClient()
    tcpClient.connect(config.port, config.host)
  }

  // disconnect from the device when stop()
  function stop() {
    tcpClient.end()
  }

  // send a 'power' command
  function setPower(params) {
    if (params.Status === "On") {
      tcpClient.write('!Power ON\n')
    } else if (params.Status === "Off") {
      tcpClient.write('!Power OFF\n')
    }
  }

  // publish public functions
  return {
    setup: setup,
    start: start,
    stop: stop,
    setPower: setPower
  }
}
```

AFTER having added the 'Set Shutter' command

```
"use strict"

// exported init function
let host
exports.init = function (_host) {
  host = _host
}

// exported createDevice function
exports.createDevice = function (base) {
  const logger = base.logger || host.logger
  let config
  let tcpClient

  function setup(_config) {
    config = _config
  }

  // connect to the device when start()
  function start() {
    initTcpClient()
    tcpClient.connect(config.port, config.host)
  }

  // disconnect from the device when stop()
  function stop() {
    tcpClient.end()
  }

  // send a 'power' command
  function setPower(params) {
    if (params.Status === "On") {
      tcpClient.write('!Power ON\n')
    } else if (params.Status === "Off") {
      tcpClient.write('!Power OFF\n')
    }
  }

  // CAUTION! NEW DRIVER 'Set Shutter' COMMAND ADDED!!!!
  // send a 'shutter' command
  function setShutter(params) {
    if (params.Status === "On") {
      tcpClient.write('!Shutter ON\n')
    } else if (params.Status === "Off") {
      tcpClient.write('!Shutter OFF\n')
    }
  }

  // publish public functions
  return {
    setup,
    start,
    stop,
    setPower,
    setShutter, // CAUTION! Don't forget to publish the 'Set Shutter' command!!!
  }
}
```

10.3.2 Forgetting to start the polling

The Polling Manager is a very convenient way to manage device polling. But be aware that it doesn't start polling automatically when the device starts. This has to be done manually (usually in the `start()` function).

Note however that the Polling Manager automatically stops pollings when the device is stopped.

```
function start() {
  base.startPolling()
}
```

11 Device Harmonization Guidelines

This document is meant to help develop drivers that are in line with the Barco Overture Device Standards.

These standards exist to help harmonize similar type of devices so that commands and variables are consistent. This allows external interfaces and designs to take advantage of how to interact with the devices.

11.1 Reasons To Harmonize

Harmonzing drivers has advantages when programming new systems. For example, most projectors can be turned on and off. Different models use different protocols to achieve this, which is why we have different drivers for different models. If the GUI also has to have a different way to talk to each driver, then you spend extra time programming when the functionality is the same no matter what driver is being used.

Another advantage is being able to devices for new make and models, without effecting the current interfaces.

Interfaces also gain an advantage when they know more about the data they are dealing with. Interacting with "Temperature" variables that are always real type variables, allows the interface to format it easily and display in a more readable manner.

11.2 Different Functionalities

Standardizing devices does not mean every device must act the same. Devices in the real world differ in functionality between make and model. For example, some projectors have shutters, others do not. In this scenario, if a projector does not have shutter control there is no reason to force the driver to have shutter control. Barco Overture accounts for different functionalities by having different variables. If a Projector driver has a "Shutter" variable, the interface knows it can interact with it.

The driver must have this variable, even if the device is not capable of properly knowing it's state. If the device does not provide proper feedback for it's state, the driver must internally do so or must at least provide the variable even if it's state is not truly known.

This contract of having a variable for a functionality is not always one to one. For example, a player may have the ability to pause while others may not. You do not need a "Pausing" variable, however in this case you must tell the "TransportStatus" variable that he has a "Paused" state. In this way, the driver can still tell the interface what functionalities he has.

11.2.1 Deleting Functionalities

The standards provided in this document, will not always match the functionalities of your device. This document will have some functionalities that your device might not be able to handle. In these cases do not publish the associated variables. Another example, if your display does not have audio functionalities, do not publish the audio variables. The interface will know not to provide access to the audio control functionalities.

11.2.2 Adding Functionalities

At the same time, your device might have special functionalities not available to most other devices of it's type. For example, if your projector has audio support you may want to have control. In these scenarios, you are free to add as many commands/variable contracts as you wish to define new functionalities. These contracts should try to use exisiting contracts from other devices where applicable. In the above example, because other devices have audio capabalities, the projector's audio functionalities should use those commands and variables for it's own audio functionalities.

If the functionalities you are writing are completely new to any device type, define them as you wish using the contract guidlines provided in this document.

11.3 Barco Overture Interoperability Chart

This document gives guide lines to implement Barco Overture Interoperability Chart for devices. However, MIC does not limit itself to devices, indeed, it may be applied to other entities whenever a gobal interoperable model is required.

Within Overture, but more generally in any automation application, automatically generated GUIs and "feature based" automated tasks cannot be achieved when dealing with different equipent's type/model/behavior.

Having a common device model within Overture which standardizes the commands/variables set as per device class basis, is a keypoint to achieve this goal. MIC will be supported by the underlying OpenCap protocol (whether on its xml or json implementation).

11.4 General guide lines and notes

- MIC contains a standardized variables & commands set per device class or category
- Any new created driver supporting a device within exisiting class should implement MIC
- Commands/variables skeleton or templates such as (room, zone, projector, etc...) should be provided as par of MIC
- MIC has a version which define its specifications such as list of device class and command/variable sets. See the first page (top of the document) for the current version
- Not all commands/variables must be implemented in a specific MIC device type, depending on the actual device capabilities. The general rule is that each command has an associated variable which reflects the state which that command controls. If this variable is not publihed, it is assumed that the associated command is not available.
  - Ex: `Set Shutter` command and `Shutter` variable.
  - Ex: `Set Brightness` command and `Brightness` variable
- For existing drivers such as MXMs, see Existing MXMs approach
- For audio levels, in order to accomodate with devices which have only one direction of audio (i.e most of the time, audio out) and devices which have bi-directionnal audio (audio output and audio input), the post-fix of `In` will be applied for inputs whereas `Out` won't be applied for output; in other words, audio default to output (Ex: `Set Audio Level` and `Set Audio Level In`).

11.5 MIC Commands/Variables set

The term 'device' is used here as a generic term to define an entity which has a command set and a variable set. One can argue that a room or a floor is not a device but in an automation standpoint, powering on a room has the same semantic as powering on a video projector. Similarly, the functional status of a room or a video projector have also the same semantic.

Each command, parameter or variable is identified by its name (as a sequence of characters). It is then important that those names are unique within a device class. The identification of a command, parameter or variable name is done either using its full name such as 'Video Player' or 'Set Level' or its compact name such as `videoplayer` or `setlevel`, (see Compact Names for more details. Consequently, the unicity must also be preserved for compact names as well.

11.5.1 Compact Names

A compact name is a name more designed to be used programmatically. A compact name is basically a full name but without any space or special character, all letters are lowercase. For examples:

- 'Video Player' compact name is `videoplayer`
- 'Get Preset List' compact name is `getpresetlist`
- 'Temperature Set Point' compact name is `temperaturesetpoint`
- 'Current Channel Level - 1' compact name is `currentchannellevel1`

11.6 Devices

Current supported device class are:

- Light
- HVAC
- Power
- Player (Audio/Video)
- Audio System
- Projector
- Display
- A/V Conference
- Camera
- Matrix
- Lift
- Shades
- I/O
- Keypad

11.6.1 Light

Overture subtype ID: `lighting`.

11.6.1.1 Commands

- Set Power (Channel: *Integer*, Status : *Enum:[Off/On]*)
- Set Level (Channel: *Integer*, Level: *Real*)
  - Level parameter is indicated in % by default
- Recall Preset (Name: *String*)

11.6.1.2 Variables

- Status (*Enum: [Disconnected/Connected]*)
- Power - per Channel (*Enum: [Off/On]*)
- Level - per Channel (*Real*)
  - Level parameter is indicated in % by default
- Presets (*Enum: [None/Based On Device]*)
- ChannelCount (*Integer*)

11.6.2 HVAC

Overture subtype ID: `hvac`.

11.6.2.1 Commands

- Set Temperature (Zone: *Integer*, Temperature: *Real*)
  - Zone number is set to 0 if there is no zone
  - Level parameter is indicated in degree whether degree C or degree F depending on the actual device settings
- Set Occupancy Mode (Zone: *Integer*, Status: *Enum: [Auto / Manual]]*)
  - Zone number is set to 0 if there is no zone
- Set Occupancy Status (Zone: *Integer*, Status: *Enum: [Unoccupied / Occupied]*)

- Zone number is set to 0 if there is no zone

11.6.2.2 Variables

- Status (*Enum: [Disconnected/Connected]*)
- OccupancyMode - per Zone (*Enum: [Auto / Manual]*)
- OccupancyStatus - per Zone (*Enum: [Unoccupied / Occupied]*)
- CurrentTemperature - per Zone (*Real*)
- TargetTemperature - per Zone (*Real*)

11.6.3 Power

Overture subtype ID: `power`.

11.6.3.1 Commands

- Set Power (Channel: *Integer*, Status : *Enum:[Off/On]*)

11.6.3.2 Variables

- Status (*Enum:[Disconnected/Connected]*)
- Power - per Channel (Status : *Enum:[Off/On/Powering Off/Powering On]*)
- ChannelCount (*Integer*)

11.6.4 Player (Audio/Video)

Overture subtype ID: `player`.

11.6.4.1 Commands

- Play
- Pause
- Stop
- Rewind
- Forward
- Previous
- Next
- Record
- Load Media (Media: *String*)
- Locate (Position: Time)

11.6.4.2 Variables

- Status (*Enum:[Disconnected/Connected]*)
- TransportStatus (*Enum:[Stopped/Playing/Paused/Rewinding/Forwarding/Previous/Next/Recording]*)
- Position (Time)
- CurrentMedia (*String*)

11.6.5 Audio System

Overture subtype ID: `audiosystem`. Simple audio system may be restrain to audio level according to the feature list.

11.6.5.1 Commands

- Set Audio Level (Channel: *Integer*, Level: *Real*)
  - Level parameter is indicated in % by default
- Set Audio Level In (Channel: *Integer*, Level: *Real*)
  - Level parameter is indicated in % by default
- Recall Preset (Name: *String*)
- Select Source (Channel: *Integer*, Name: *String*)
- Set Bass (Channel: *Integer*, Level: *Real*)
  - Level parameter is indicated in % by default
- Set Treble (Channel: *Integer*, Level: *Real*)
  - Level parameter is indicated in % by default
- Set Audio Mute (Channel: *Integer*, Status: *Enum:[Off/On]*)
- Set Audio Mute In (Channel: *Integer*, Status: *Enum:[Off/On]*)

11.6.5.2 Variables

- Status (*Enum:[Disconnected/Connected]*)
- Presets (*Enum: [None/DynamicList]*)
- Sources - Per Output (*Enum: [Based On Device]*)
- AudioLevelInput - Per Input (*Real*)
  - Level parameter is indicated in % by default
- AudioMuteInput - Per Input (Status: *Enum:[Off/On]*)
- AudioLevel - Per Output (*Real*)
  - Level parameter is indicated in % by default
- AudioMute - Per Output (Status: *Enum:[Off/On]*)
- Bass - per Channel (*Real*)
  - Level parameter is indicated in % by default
- Treble - per Channel (*Real*)
- Level parameter is indicated in % by default

11.6.6 Projector

Overture subtype ID: `projector`.

11.6.6.1 Commands

- Set Power (Status: *Enum: [Off/On]*)
- Set Video Mute (Status: *Enum: [Off/On]*)
- Select Source (Name: *String*)
- Set Video Freeze (Status: *Enum: [Off/On]*)
- Set Brightness (Level: *Real*)
  - Level parameter is indicated in % by default
- Set Contrast (Level: *Real*)
  - Level parameter is indicated in % by default
- Set Focus (Position: *Real*)
  - Level parameter is indicated in % by default
- Set Shutter (Status: *Enum: [Closed/Open]*)
- Set Zoom (Position: *Real*)
  - Position parameter is indicated in % by default

11.6.6.2 Variables

- Status (Status: *Enum:[Disconnected/Connected]*)
- Power (Status: *Enum:[Off/On/Powering Off/Powering On]*)
- Sources (*Enum: [Based On Device]*)
- VideoMute (Status: *Enum:[Off/On]*)
- VideoFreeze (Status: *Enum:[Off/On]*)
- Shutter (*Enum: [Closed/Open]*)
- HoursLamp - per Lamp (*Real*)
- Temperature (*Real*)
- Brightness (*Real*)
  - Level parameter is indicated in % by default
- Contrast (*Real*)
  - Level parameter is indicated in % by default
- Focus (*Real*)
  - Level parameter is indicated in % by default
- Zoom (*Real*)
  - Position parameter is indicated in % by default

11.6.7 Display

Overture subtype ID: `display`.

11.6.7.1 Commands

- Set Power (Status: *Enum[Off/On]*)
- Set Audio Mute (Status: *Enum[Off/On]*)
- Set Audio Level (Level: *Real*)
  - Level parameter is indicated in % by default
- Select Source (Name: *String*)
- Set Brightness (Level: *Real*)
  - Level parameter is indicated in % by default
- Set Contrast (Level: *Real*)
  - Level parameter is indicated in % by default

11.6.7.2 Variables

- Status (Status: *Enum:[Disconnected/Connected]*)
- Power (Status: *Enum:[Off/On/Powering Off/Powering On]*)
- Sources (*Enum: [Based On Device]*)
- AudioMute (Status: *Enum:[Off/On]*)
- AudioLevel (*Real*)
  - Level parameter is indicated in % by default
- Temperature (*Real*)
- Brightness (*Real*)
  - Level parameter is indicated in % by default
- Contrast (*Real*)
  - Level parameter is indicated in % by default

11.6.8 Audio/Video Conference

Overture subtype ID: `avconference`.

11.6.8.1 Commands

- Call Number (Number: *String*)
- Call Name (Name: *String*)
- Hang Up
- Answer
- Set Audio Level (Output: *Integer*, Level: *Real*)
  - Level parameter is indicated in % by default
- Set Audio Level In (Input: *Integer*, Level: *Real*)
  - Level parameter is indicated in % by default
- Set Audio Mute In (Input: *Integer*, Status: *Enum: [Off/On]*)
- Set Audio Mute (Output: *Integer*, Status: *Enum: [Off/On]*)
- Set Video Mute (Status: *Enum: [Off/On]*)
- Key Press (Name: *String*)

11.6.8.2 Variables

- Status (Status: *Enum:[Disconnected/Connected]*)
- AddressBook (*Enum: [None/*DynamicList*]*)
- Number (*String*)
- AudioLevelInput - Per Input (*Real*)
  - Level parameter is indicated in % by default
- AudioLevel - Per Output (*Real*)
  - Level parameter is indicated in % by default
- AudioMuteInput - Per Input (Status: *Enum: [Off/On]*)
- AudioMute - Per Output (Status: *Enum: [Off/On]*)
- VideoMute (Status: *Enum:[Off/On]*)

11.6.9 Camera

Overture subtype ID: `camera`.

11.6.9.1 Commands

- Set Power (Status: *Enum: [Off/On]*)
- Recall Preset (Name: *String*)
- Set Video Mute (Status: *Enum: [Off/On]*)
- Set Pan (Position: *Real*)
  - Position parameter is indicated in degrees by default
- Set Tilt (Position: *Real*)
  - Position parameter is indicated in degrees by default
- Set Zoom (Position: *Real*)
  - Position parameter is indicated in % by default
- Set Focus (Position: *Real*)
  - Position parameter is indicated in % by default

11.6.9.2 Variables

- Status (*Enum:[Disconnected/Connected]*)
- Power (*Enum:[Off/On/Powering On/Powering Off]*)
- Presets (*Enum: [None/*DynamicList*]*)
- VideoMute (Status: *Enum:[Off/On]*)

- Pan (*Real*)
  - Position parameter is indicated in degrees by default
- Tilt (*Real*)
  - Position parameter is indicated in degrees by default
- Zoom (*Real*)
  - Position parameter is indicated in % by default
- Focus (*Real*)
  - Position parameter is indicated in % by default

11.6.10 Matrix

Overture subtype ID: `matrix`.

11.6.10.1 Commands

- Recall Preset (Name: *String*)
- Select Source (Channel: *Integer*, Name: *String*)

11.6.10.2 Variables

- Status (*Enum:[Disconnected/Connected]*)
- Presets (*Enum: [None/*DynamicList*]*)
- Sources - per Output (*Enum: [Based On Device]*)

11.6.11 I/O

Overture subtype ID: `io`.

11.6.11.1 Commands

- Set Digital Output - per Channel (Channel: *Integer*, Status : *Enum:[Off/On]*)
- Set Analog Output - per Channel (Channel: *Integer*, Level: *Real*)

11.6.11.2 Variables

- Status (*Enum:[Disconnected/Connected]*)
- DigitalInput - per Channel (Status : *Enum:[Off/On]*)
- DigitalOutput - per Channel (Status : *Enum:[Off/On]*)
- AnalogInput - per Channel (Level: *Real*)
- AnalogOutput - per Channel (Level: *Real*)

11.6.12 Lift

Overture subtype ID: `lift`.

11.6.12.1 Commands

- Move (Status: *Enum: [Stop/Up/Down]*)
- Stop

11.6.12.2 Variables

- Status (*Enum:[Disconnected/Connected]*)
- Position (*Enum:[Stopped/Up/Down/Moving Up/Moving Down]*)

11.6.13 Shades

Overture subtype ID: `shades`.

11.6.13.1 Commands

- Move (Status: *Enum: [Stop/Close/Open]*)
- Rotate (Status: *Enum: [Stop/Close/Open]*)
- Stop

11.6.13.2 Variables

- Status (*Enum:[Disconnected/Connected]*)
- Position (*Enum:[Stopped/Closed/Opened/Closing/Opening]*)
- Rotation (*Enum:[Stopped/Closed/Opened/Closing/Opening]*)

11.6.14 Keypad

Overture subtype ID: `keypad`.

Can be used for regular keypads.

This device has not much standard commands or variables apart from the usual `Status`.

11.6.14.1 Variables

- Status (*Enum:[Disconnected/Connected]*)

11.7 Extra Types

Aside from devices, certain types can be standardized to take advantage of harmonization

- Room
- Zone
- Floor
- Building

11.7.1 Room

11.7.1.1 Commands

- Recall Preset (Name: *String*)
- Set Power (Status: *Enum: [Off/On]*)
- Set Occupancy Mode (Zone: *Integer*, Status: *Enum: [Auto / Manual]]*)

    - Zone number is set to 0 if there is no zone
- Set Occupancy Status (Zone: *Integer*, Status: *Enum: [Unoccupied / Occupied]*)
    - Zone number is set to 0 if there is no zone

11.7.1.2 Variables

- Power (Status: *Enum:[Off/On/Powering On/Powering Off]*)
- Presets (*Enum: [None/*DynamicList*]*)
- OccupancyMode - per Zone (*Enum: [Auto / Manual]*)
- OccupancyStatus - per Zone (*Enum: [Unoccupied / Occupied]*)

11.7.2 Zone

11.7.2.1 Commands

- Recall Preset (Name: *String*)
- Set Power (Status: *Enum: [Off/On]*)
- Set Occupancy Mode (SubZone: *Integer*, Status: *Enum: [Auto / Manual]]*)
    - SubZone number is set to 0 if there is no zone
- Set Occupancy Status (SubZone: *Integer*, Status: *Enum: [Unoccupied / Occupied]*)
    - SubZone number is set to 0 if there is no zone

11.7.2.2 Variables

- Power (Status: *Enum:[Off/On/Powering On/Powering Off]*)
- Presets (*Enum: [None/*DynamicList*]*)
- OccupancyMode - per SubZone (*Enum: [Auto / Manual]*)
- OccupancyStatus - per SubZone (*Enum: [Unoccupied / Occupied]*)

11.7.3 Floor

11.7.3.1 Commands

- Recall Preset (Name: *String*)
- Set Power (Status: *Enum: [Off/On]*)

11.7.3.2 Variables

- Power (Status: *Enum:[Off/On/Powering On/Powering Off]*)
- Presets (*Enum: [None/*DynamicList*]*)

11.7.4 Building

11.7.4.1 Commands

- Recall Preset (Name: *String*)
- Set Power (Status: *Enum: [Off/On]*)

11.7.4.2 Variables

- Power (Status: *Enum:[Off/On/Powering On/Powering Off]*)
- Presets (*Enum: [None/*DynamicList*]*)

11.8 Functionality Guidelines

When setting up a new functionality for a device, there are a certain set of guidelines to follow to help with harmonization.

11.8.1 Commands/Parameters

When designing a driver with a new functionality, first check to see if the functionality exists in another device.

For example, a projector that has audio options should use Set Audio Mute (Status: *Enum[Off/On]*) and Set Audio Level (Level: *Real*) as it's commands. This way, interfaces interacting with the projector will inhertly know how it works.

If the functionality you are designing is not within an existing device, you will be able to create the command as you need. Use exisiting commands as guidlines for naming and structure.

For example, if you are designing a functionality that rotates a projector left or right. You should create only one command for the functionality. A command like 'Set Rotation' with a parameter is better then two commands 'Rotate Left'/'Rotate Right'.

Command parameters should also be standardized between devices whenever possible. Instead of using the the exact name of the item being controlled, use general names so the command can be adapted to other devices For example, when creating a command to control a mute, instead of using Mute as a the parameter name, use Status.

By default, the following parameter names should be used where applicable:

- Status: When dealing with states of objects such as Off/On
- Level: When dealing with relative values such as %. This can also be applied to absolute values that might otherwise be relative such as audio level which could be in db.
- Position: When dealing with absolute values.
- Name: When calling an object that is associated by it's name such as a preset or source.

11.8.2 Variables

When designing a new functionality a for devices, make sure the functionality is represented by a variable. Interfaces rely on variables to know what functionalites they have.

Just as with commands, when a functionality is similar to another device, use the exisiting variable names to show this functionality.

If a new variable is needed, first determine if the variable is only used once per device or is a per X type of variable (for example per channel or per lamp).

The type should be based on how the variable will be used. For example, if the variable is monitoring current known state within a list of known states an enum maybe appropiate. If monitoring a state, when the known states could be infinite a string maybe more appropiate.

When dealing with per X type variables, naming convention should be in a format that determines what your monitoring first, then the per item, and then the index. For example when monitoring lamp hours, we write HoursLamp1, HoursLamp2, etc. For filtering, we can now easily find all lamp hours because the string starts with HoursLamp.

11.9 Driver Design

Aside from harmonization commands and variables, drivers can be given extra info so that interfaces know how to interact with them.

11.9.1 Parameters

11.9.1.1 Min/Max

The range of parameter's value might has to be specified to insure harmonization between devices.

For example, the min and max brightness values for a projector are likely different depending on the make/model.

Therefore, a command asking for a certain value for a projector might not work (or have the same effect) with another projector.

12 Revisions

12.1 Overture CS 1.7.0

- Improved: IVariable Interface supports a new function: `getLastChangedTime()`
- Improved: In the Logs tab, re-enabling the Logs Activity does not clear the logs anymore.
- Added: In the Server tab, double clicking on Utilities will display few seconds a button allowing to download the project.
- Added: A "Secure communication to download drivers" icon is displayed in the header.
- Fixed: When the ControlServer has no password, each time the UXServer modifies its project, the GUI goes back to the Server tab instead of staying on the selected tab.
- Fixed: Logs were not stored anymore while the Logs were stopped.
- Fixed: Devices tab reverts back to the first device in the list each time a project is received.

12.2 Overture CS 1.6.0

- Added: A password protection has been added to the ControlServer. If a password has been defined in Configurator for a ControlServer, this one will show a login page on the GUI and avoid any other action while the user is not authenticated.
- Fixed: An unhandled exception error was logged when the limit of commands in the queue is reached by a device.
- Fixed: Perform command using filter were executed on every points.
- Improved: remote perform command now support some extra options IPerformOptions Interface.
- Fixed: In the 'Devices' tab of the ControlServer GUI, the selected device went back to the first device in the list each time a project was received.

12.3 Overture CS 1.5.2

- Added: QA Check Points and Driver Quality Checklist chapters have been added to the sdk documentation.
- Fixed: The `Evaluate the expression on start` did not work in the `Trigger` behavior (advanced options).
- Fixed: When the `timeBetweenCommands` option is used, this delay is not always respected.
- Fixed: Scrolling the Logs view was slow because long traces where present. These logs are now truncated.
- Fixed: Using `NetworkUtilities`, the ping function might throw an error depending how the answer is formatted.
- Modified: The Lift/Shades harmonization have been revisited.
- Improved: The `Command not found` error is now logged only once, when and if it's needed.

12.4 Overture CS 1.5.1

- Added: Support inter CX. It means that a CX is now able to send a perform command or set a variable to another CX (even using filters). A CX can also be notified when a variable from another CX changes.
- Added: Support Enum labels.
- Fixed: Connection status with the UX was not properly updated.

12.5 Overture CS 1.4.1

- Added: `precision` parameter for variables of type `real`.
- Fixed: `Warn` logs are not visible in the ControlServer GUI unless the text is selected.
- Fixed: Cannot modify Real variable using sliders on the CX GUI.

12.6 Overture CS 1.4.0

- Added: Support Behaviors 2.0.
- Added: When the environment variable named `NODE_ENV` equals `development`, drivers are loaded without checking the `overtureCSVersion` property.
- Added: An anti-larsen mechanism has been added in order to break loops related to variable changes which generate a high CPU usage. it stops changing the variable value if this one changes too fast during a specific period. The period is defined through an environment variable named `VARCHANGELOOPBREAKER_TIMEFRAME` (1000 ms by default) and the admissible change count is set by `VARCHANGELOOPBREAKER_CHANGECOUNT` (20 changes by default). These parameters depend of the system where the ControlServer is installed and might require an adaptation.
- Improved: The Control Server uses NodeJS 8.
- Fixed: Some driver functions like `getSWVersion` associated to a command are no more executed by the OvertuceCS 1.3.1 version. Note: A driver function name linked to a command must respect the 'camel cased' format.
- Improved: Tasks are now supporting a condition per cue.
- Modified: `LOGLEVEL` environment variable has been uncoupled from the Logs Settings accessible in the Overture CS GUI (see ILogger Interface).

12.7 Overture CS 1.3.1

- Added: `setCommandManagerOptions(options:ICommandManagerOptions)` function has been added to device base. It allows to define a delay between commands.
- Improved: `addVariableListener()` and `removeVariableListener()` functions have been added to IBaseDevice Interface. It's recommanded to use them instead of IHost interface because listeners are automatically removed when the device is stopped. Note: only the started devices are notified if a variable changes.
- Added: `setTimeout`, `clearTimeout`, `setInterval`, `clearInterval` functions have been added to IBaseDevice Interface.
- Added: `destroy()` and `onProjectLoaded()` are optional functions added to the `IDevice` interface.
- Added: `ITCPClientOptions` has been added to the ITcpClient Interface. It modifies the previous TCPClient behavior. Now, by default, if nothing has been received since 1 minute, the TCPClient will disconnect and automatically reconnect.
- Fixed: the ControlServer was not able to download drivers when the UXServer URL is finishing by `/`.
- Improved: Logs related to a device (ex: when a device's variable changes) are no more traced by `Core` but directly by the device itself.

12.8 Overture CS 1.3.0

- Added: overhauled logging system: general, per driver or per device. Configurable from the OvertureCS GUI.
- Improved: For a driver, the `overtureCSVersion` property should be added in the 'overture' node of the package.json (instead of the root, which is deprecated) to specify the miminum Overture CS version able to supprot the driver.
- Added: `startPolling()` function accepts an `options` parameter of type `IStartPollingOptions`.
- Added: `Start Devices` / `Stop Devices` buttons are now available in the Devices view and allow to start or Stop all devices.
- Added: `setPoll(options:IPollingRequestOptions)` function has been added to device base. It provides a way to directly execute the polling request when `startPolling()` is called.

- Added: The `Reconnect` button in the GUI Server page allows to Re-initialize the connection with the UX server and reload the project.
- Added: The `Restart` button in the GUI Server page allows to Restart OvertureCS.
- Modified: Harmonization of `Audio System` has been modified, `SourcesOutput` variable has been renamed to `Sources` and the `ChannelCount` variable is no more mandatory.
- Fixed: If OvertureCS is able to establish a connection with UXServer during starting, it should not load the backuped project file but directly the project coming from the UXServer.
- Added: `NetworkUtilities` is now available. see INetworkUtilities Interface

12.9 Overture CS 1.2.0

- Added: `addVariableListener()` and `removeVariableListener()` functions have been added to the IHost interface.
- Fixed: An issue can happen when a driver is downloaded and installed.
- Added: A button is now available in the Overture CS interface to secure or not the communication to download drivers.
- Improved: A driver device function is not called when the command parameters don't match with its description.
- Fixed: Potentially, a wrong url can be opened (as a CS GUI) when the Overture CS is re-installed on windows.
- Added: The Overture CS GUI displays the logs.
- Added: An `overtureCSVersion` property can be added in the root of the package.json of a driver to specify the miminum Overture CS version able to support the driver.
- Added: `createVariable()` function has been added to the IBaseDevice Interface.
- Improved: IVariableConfig interface has been modified in order to support metadata like `icon`, `widget`...

12.10 Overture CS 1.1.0

- Fixed: `perform` command and `set variable value` request, coming from the UXSever, only accept index of enum and not its value as parameter (or the opposite, depending of the driver).
- Added: It's now possible to define an `alias` for a driver command. This feature is mainly used for commands like `start` or `stop`.
- Added: Log traces have been added when a device cannot be loaded.
- Fixed: Overture CS GUI widgets are not reset if no change on the variable's value happens.
- Fixed: Overture CS doesn't handle UXServer Url ending with "/", neither when UXServer Url is not defined.

12.11 Overture CS 1.0.0

- Initial