OpenSync™

OpenSync Overview

Date: August 10, 2020
Document ID: EUB-020-013-001

Table of Contents

References

[1] EUB-020-033-101 OpenSync 2.2 Northbound API

[2] EUB-020-033-102 OpenSync 2.0 Southbound API

[3] EIN-020-032-501 Adding new managers to OpenSync 2.0

[4] EUB-020-060-301 OpenSync Service Portfolio

Introduction

OpenSync™ is a cloud-agnostic open-source software for the delivery, curation, and management of wireless services for the home. The *OpenSync* software is built on top of several silicon vendor SDKs. It supports additional, vendor-specific 3rd party apps, and allows for customization by modifying core functionality.

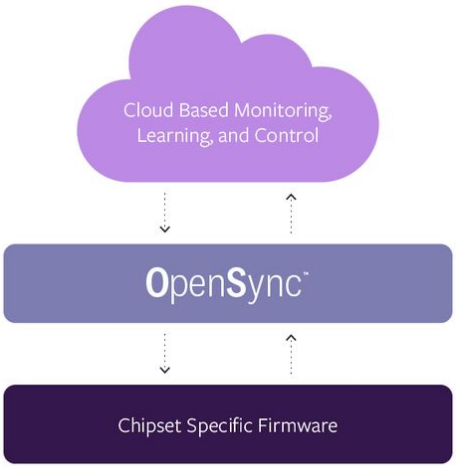


Figure 1: *OpenSync overview*

Description

OpenSync core functionality includes:

- Tunable logging infrastructure that covers all of the software Components
- Establishing and maintaining cloud connectivity via IPv4 or IPv6
- Statistics gathering, processing, and reporting for things like:
    - CPU and Memory usage
    - Associated client statistics
    - Channel survey statistics
    - Neighboring AP statistics
    - Non-associated client statistics
- Virtualized network and Wireless management
- Synchronizing WiFi settings such as SSID and Password to the cloud to be used for propagation to mesh extenders
- Managing WiFi settings such as channel number and channel width
- Setting up data path to WiFi extenders, including multiple VLAN support
- Performing steering actions

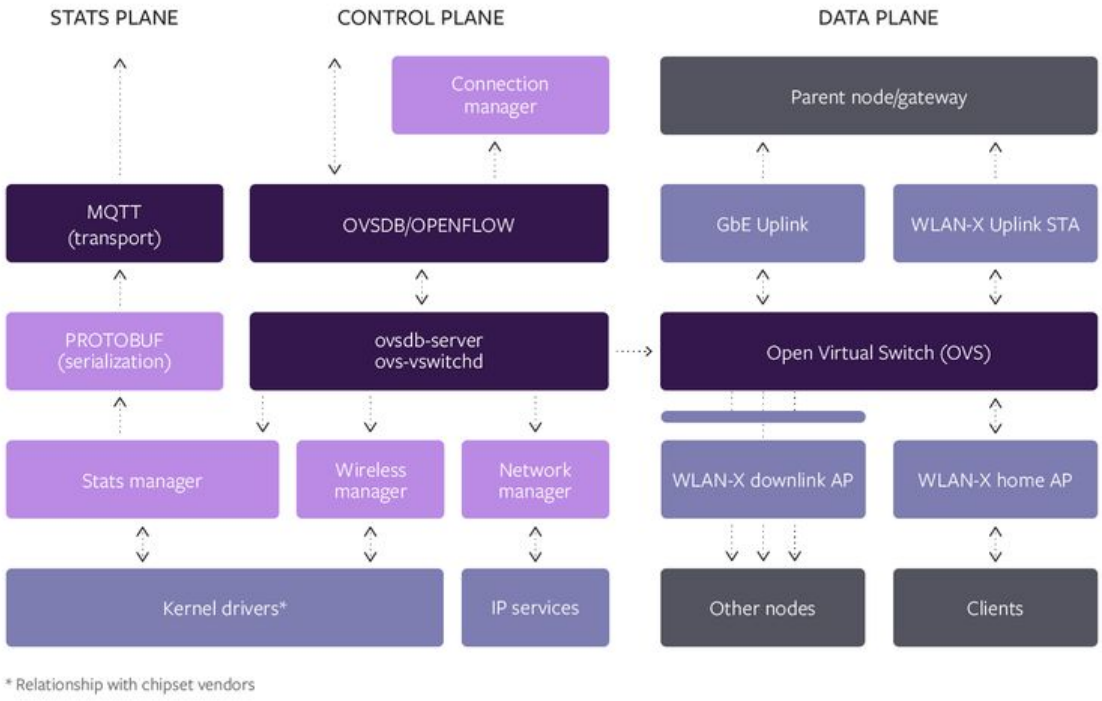


Figure 2: *OpenSync plane diagram*

OpenSync advanced features rely heavily on **OpenVSwitch** (OVS) and **Mosquitto** libraries for core functionality. Mosquitto is a message broker, implementing MQTT messaging protocol. *OpenSync* component called Stats Manager (SM) collects statistics and uses mosquitto to send data to the cloud. Two components of OpenVSwitch enable *OpenSync* to operate: "ovs-vswitchd" is the OVS daemon that controls virtual switches in the software and "ovsdb-server" is the OVS database server that contains configuration and state for various parts of *OpenSync*, like the Connection Manager, Wireless Manager, and the Network Manager. The cloud and *OpenSync* are connected through OpenFlow protocol, editing entries in the ovsdb tables is the way cloud and *OpenSync* exchange system information and instructions. Both sides use a predefined schema to set configuration values and monitor status values.

Prerequisites

OpenSync is not a standalone piece of software, it is a middle layer that relies on certain prerequisites in order to provide core functionality. *OpenSync* is built on top of an SDK, usually provided by a silicon chip vendor, for the chip used in the device. There is often a need for apps, libraries and modifications to *OpenSync*, this can be done with the Target Library.

The SDK must provide the following components for *OpenSync* to build successfully:

- Cross-compiling toolchain for chip architecture
- Wi-Fi and other drivers
- Libraries:
    - libev
    - jansson
    - mosquitto (MQTT)
    - openvswitch
    - protobuf

Other packages, libraries or tools needed for a successful build depend on vendor-specific third-party applications (see target layer). *OpenSync* can be integrated and built within the platform SDK similarly to any other package or third-party application.

Additional SDK-specific components may be developed that adapt *OpenSync* to the SDK software stack third-parties required. Examples of connections to the SDK software:

- linking Clouds' logging system to SDK logging system
- SDK specific networking utilities for managing VLANs, GRETAPs, and Bridges
- device info tool for entity information
- SDK specific agent for synchronizing configuration changes
- Wifi HAL for SoC interaction such as steering, statistics, Wi-Fi management, SSID/password management, etc.

OpenSync supports both - gateways or extenders[1].

[1] Version *OpenSync* 1.0 supports only Gateways

OVSDB and Cloud Schema

The ovsdb-server on the device needs to be started with the Cloud schema for managers to work. Managers like NM2 and WM2 rely on certain ovsdb tables to be present for normal operation, so the ovsdb server needs to be started before any managers are run, this is a part of system initialization.

The main purpose of OVSDB is to serve as interprocess communication between services in the cloud and *OpenSync* managers on the device, as illustrated in the image below. Tables in the Cloud schema are roughly organized into two groups - “config” and “state” tables. Usually, the cloud writes instructions and configuration into the config tables and the device must report its status in the state tables.

During the build process, the ovsdb database file is created from the Cloud schema. This database file will be loaded by ovsdb-server on the device. Managers will expect some values in the database to be preset, this is done by hooks to the build process in form of json or shell scripts located in ovsdb directory in any of the source layers, for example:

```
vendor/vndx/ovsdb/
├── led_config.json.sh
├── bridge_home_inet.json
├── bridge_wan_inet.json
├── eth_default.json.sh
└── radio.json.sh
```

List of tables and more details regarding *OpenSync* schema can be found in *OpenSync 2.2 Northbound API* document [1].

Directory Structure

The **build** folder contains all *OpenSync* common makefiles that are not unit specific (*unit.mk*) or target specific.

The **interfaces** folder holds OVS schema and MQTT protobuf definition for device stats.

The **ovsdb** folder contains files and scripts that pre-populate the ovs database at compile time. This is necessary in order to keep information generated during the build procedure, like the Cloud redirector address, which is the first address the gateway attempts to connect to, and the version matrix, with which Cloud services identify devices.

The **src** folder contains *OpenSync* source code: **tools**, **libraries**, and **managers.** The structure of each library, manager or tool is that of a “unit”.

The **platform** folder contains platform specific code. Since *OpenSync* is supported on different platforms, code separation between platform specific (mostly SDK specific) and target specific code was implemented, since several targets can use the same platform.

The **vendor** folder follows the same logic as the platform directory. *OpenSync* works with different vendor middle layer or abstraction layer implementations, so vendor specific code is located in this directory.

This is what the *OpenSync* directory structure looks like before any build procedure was executed:

```
core
├── build
├── interfaces
├── ovsdb
├── platform -> ../platform
├── vendor -> ../vendor
└── src
    ├── bm
    ├── cm2
    ├── cpm
    ├── dm
    ├── fcm
    ├── fsm
    ├── fut
    ├── hello_world
    ├── lib
    ├── lm
    ├── nfm
    ├── nm2
    ├── om
    ├── pm
    ├── qm
    ├── sm
    ├── tools
    ├── um
    ├── wano
    ├── wm2
    └── xm
```

During the build process additional folders are created: images and work.

The **images** folder contains *OpenSync* build artifacts or SDK build artifacts, like the image binary or rootfs tarball.

The **work** folder is elsewhere sometimes described as “build directory”, it contains subfolders for each build target (specified by “TARGET=target1” variable) - several may be built in the same workspace - and each of these subfolders contains all object files, build binaries and libraries and rootfs. For more details, see the chapter Building.

The structure looks like this:

```
core
├── build
├── images
├── interfaces
├── ovsdb
├── platform -> ../platform
├── vendor -> ../vendor
├── src
│      ├── bm
│      ├── cm2
│      ├── cpm
│      ├── dm
│      ├── fcm
│      ├── fsm
│      ├── fut
│      ├── hello_world
│      ├── lib
│      ├── lm
│      ├── nfm
│      ├── nm2
│      ├── om
│      ├── pm
│      ├── qm
│      ├── sm
│      ├── tools
│      ├── um
│      ├── wano
│      ├── wm2
│      └── xm
│
└── work
        ├── target1
        └── target2
```

Building

To build *OpenSync*, go to core repo and use:

```
make TARGET=device-model
```

This is usually run by a "recipe" or a "package" of the platform SDK.

If TARGET is unspecified then it is "native" and binaries are built for the host architecture.

The build system is composed of *units* and *layers*.

Units

Building a single unit can also be specified with the unit path, for example:

```
make TARGET=example src/lib/ds
```

Units are defined by a unit.mk file and defines the type (lib, bin), sources, dependencies, and other options.

Sample: src/lib/ds/unit.mk

```
################################################################################
# Data structures: Linked list, double linked lists, red-black trees...
################################################################################
UNIT_NAME := ds
UNIT_TYPE := LIB
UNIT_SRC += src/ds_tree.c
UNIT_CFLAGS := -I$(UNIT_PATH)/inc
UNIT_EXPORT_CFLAGS := $(UNIT_CFLAGS)
```

Layers

Layers are in most cases these: core, platform/$PLATFORM, vendor/$VENDOR, but more could be included. Each layer can add more units, or modify existing units via an *override.mk* file. For example if layer *vendor/vndx* is included then the unit definition in *src/lib/log/unit.mk* can be modified by *vendor/vndx/src/dm/override.mk*:

```
################################################################################
# DM overrides for vendor/vndx
################################################################################
# usual DM sources, except dm_hook.c
UNIT_SRC := $(filter-out src/dm_hook.c,$(UNIT_SRC))
# dm_hook.c source, that we want to override
UNIT_SRC_TOP += $(OVERRIDE_DIR)/src/dm_hook.c
```

The accompanying file *vendor/vndx/src/dm/src/dm_hook.c* would look something like this:

```
#include "dm.h"

bool dm_hook_init(struct ev_loop *loop)
{
    (void)loop;
    printf("This is an overridden function.\n");
    return true;
}

bool dm_hook_close() {
    return true;
}
```

TARGET specific configuration happens in $LAYER/build/target-arch.mk makefile. Initially, *target-arch.mk* makefiles for all layers are loaded in this order:

- core/build/target-arch.mk

- platform/*/build/target-arch.mk
- vendor/*/build/target-arch.mk

And each needs to check if TARGET is handled by that specific layer and then set additional variables such as PLATFORM, VENDOR, ARCH_MK, and a list of units that are enabled/disabled.

If PLATFORM or VENDOR variables are specified, they are added to the list of layers for which all the units and overrides are processed, otherwise, these layers are ignored. Additional layers can be specified with INCLUDE_LAYERS variable. ARCH_MK is an additional makefile, usually architecture specific, that is included in the build process.

Example: core/build/target-arch.mk handles TARGET=native

```
# default target
TARGET ?= $(DEFAULT_TARGET)
# append list of all supported targets
OS_TARGETS += native
ifeq ($(TARGET),native)
ARCH = native
ARCH_MK = build/$(ARCH).mk
CPU_TYPE = $(shell uname -m)
BUILD_LOG_PREFIX_PLUME                := n
BUILD_LOG_HOSTNAME                    := n
# Disable unneeded units
UNIT_DISABLE_src/bm                   := y
UNIT_DISABLE_src/blem                 := y
UNIT_DISABLE_src/cm2                  := n
UNIT_DISABLE_src/dm                   := y
UNIT_DISABLE_src/lm                   := y
UNIT_DISABLE_src/nm2                  := y
UNIT_DISABLE_src/om                   := y
UNIT_DISABLE_src/sm                   := y
UNIT_DISABLE_src/wm2                  := y
UNIT_DISABLE_src/lib/bsal             := y
UNIT_DISABLE_src/lib/cev              := y
endif
```

Build *work* directory is located in work/$TARGET and there one can find: objects, libs, binaries, rootfs.

```
work/
└── native-ubuntu16.04-x86_64
    ├── bin
    ├── lib
    ├── obj
    └── rootfs
```

When building, some of the goals that can be specified are:

- **all:** (default goal) build software (binaries, libs) and install it to *work/rootfs*
- **ovsdb-create:** create the initial ovs database and install it to *work/rootfs*
- **rootfs-prepare:** copy additional files (from *$LAYER/rootfs*) to *work/rootfs*
- **rootfs:** all of the above (all, ovsdb-create and rootfs-prepare)
- **rootfs-install:** all of the above plus install the work rootfs to SDK rootfs

Kconfig

Kconfig is a configuration system introduced by the Linux kernel. It is used in *OpenSync* to reduce the number of device specific #ifdefs in the code. It allows you to configure various parameters with a text mode menu interface or by directly editing the per-device config file.

Prerequisites

Installing kconfiglib:

```
# pip install kconfiglib
```

Kconfig file location

Default location is: core/kconfig/targets/config_$(TARGET)

For the vendor specific devices, this should be changed in vendor/VENDORX/build/target-arch.mk like this:

```
KCONFIG_TARGET     ?= $(VENDOR_DIR)/kconfig/targets/$(TARGET)
```

If a Kconfig file is not found, a default is used: core/kconfig/targets/config_default

Kconfig parameter definition

The Kconfig parameters are defined in a series of Kconfig files scattered across various source units. The main file is core/kconfig/Kconfig which also includes:

- platform/*/kconfig/Kconfig
- vendor/*/kconfig/Kconfig

Any additional vendor specific options can be defined there.

Changing configuration

To create or modify a configuration, use this command:

```
# make TARGET=MODELX menuconfig
```

When done, make the appropriate selections and save the configuration. Kconfig only saves the diff from default values. This means that every time another option is added to the Kconfig default, the value is automatically used at compile time. This avoids the need to manually

regenerate the Kconfig file each time a new option is added. The full Kconfig file is generated at build time and stored in the work dir.

Using Kconfig parameters

The Kconfig parameters are available in the makefile rules and in source code as CONFIG_PARAMETER.

Examples:

```
*.c: #ifdef CONFIG_PARAMETER
*.mk: ifeq ($(CONFIG_PARAMETER),...)
```

For more information on Kconfig, see:

- https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt
- https://github.com/ulfalizer/Kconfiglib

Each *OpenSync* manager has a dedicated Kconfig configuration file located at: `src/<manager_name>/kconfig/Kconfig.managers`. Two important entries are needed in this file:

- Option to enable/disable manager compilation during build time.
- Manager startup configuration. Configures the startup process for the manager.

Manager Configuration

The Kconfig file (`kconfig/targets/<target_name>`) requires the `CONFIG_MANAGER_*_CFG` parameter for every manager. The option is parsed as a string with the ";" character used as a delimiter. The first (manager name) and second (startup option) parameters are required and have a fixed place. Others are optional and can be in any order.

1. Manager name

This is the name of the binary file for this manager.

```
CONFIG_MANAGER_<name>_CFG="<name>;false"
```

The same name must be used in the manager `unit.mk` file as

```
UNIT_NAME:=<name>
```

Based on value of CONFIG_MANAGER_<name> variable manager will be compiled or not:

```
UNIT_DISABLE:=$(if $(CONFIG_MANAGER_<name>),n,y)
```

2. Startup option

This option defines if the manager is automatically started by the Diagnostics Manager (DM) or not. The accepted values are “true” or “false”.

```
CONFIG_MANAGER_<name>_CFG="<name>;false"
```

This value is parsed into the `Node_Service` table. To manually verify the manager in runtime, run this command:

```
ovsh s Node_Services --where service==<name>
```

3. Plan B option

The Plan B option makes sure that in case of a failure, the DM restarts:

- All managers (true) through target_managers_restart
- Only this particular manager (false)

The default value for plan B is false. To enable plan B behavior, add configuration:

```
CONFIG_MANAGER_<name>_CFG="<name>;true;needs_plan_b=true"
```

4. Always restart option

If this option is set to “true”, managers should always restart, even when killed by Linux signals that usually do not trigger a restart. The default value is “false” To enable this option, add configuration:

```
CONFIG_MANAGER_<name>_CFG="name;true;always_restart=true"
```

5. Restart delay option

This option defines the duration of the manager’s restart delay in case of a crash. The delay is specified in seconds. Value 0 means default delay, while value -1 means immediate restart. To set an immediate restart, add configuration:

```
CONFIG_MANAGER_<name>_CFG="name;true;restart_delay=-1"
```

The rootfs/ directory

The rootfs directory allows you to add and maintain static files that will be installed as part of the *OpenSync* package. It is mostly used to store different scripts, config files and pre-built binary files as part of the *OpenSync* package.

The rootfs directory can contain multiple sections:

- **rootfs/common/*** - installed when given *OpenSync* layer is used
- **rootfs/kconfig/<Kconfig option>/*** - installed when given Kconfig option is selected

- **rootfs/platform/<platform>/*** - installed when given platform is selected
- **rootfs/target/<target>/*** - installed when given build target is selected
- **rootfs/hooks/*** - post and preinstall scripts

Use of rootfs/common and rootfs/target

All files placed in rootfs/common will be installed on target rootfs when a given *OpenSync* layer is used. By using rootfs/target, you may guard the files so they are installed only when a given build target is selected.

Note that target specific files may override files from rootfs/common in case they are put on the same path.

In the below case, the file “/etc/foo” from “rootfs/common” will be overwritten with “/etc/foo” from the target-specific rootfs layer in case when a given build target is selected.

```
rootfs/common/
└── etc
    └── foo
rootfs/target/
└── TARGET_ABC
    └── etc
        └── foo
```

Use of rootfs/kconfig

It is possible to install guard file installation based on the selected Kconfig options. For this, we must use the following path:

- **rootfs/kconfig/<Kconfig option>/...**

In the below example, the file “/etc/foo” will be installed only when the CONFIG_USE_FOO Kconfig option is enabled.

```
rootfs/kconfig/
└── USE_FOO
    └── etc
        └── foo
```

Use of rootfs/hooks

This directory allows you to add post-install and pre-install hooks in the form of shell scripts which get executed at the appropriate time during build.

The rootfs/hooks directory also allows you to use common or target specific pre/post-install hooks.

```
rootfs/hooks/
├── common
│   ├── post-install
│   └── pre-install
└── target
    ├── TARGET_ABC
    │   └── post-install
    └── TARGET_XYZ
        ├── post-install
        └── pre-install
```

Use of INSTALL_PREFIX

Special directory INSTALL_PREFIX is handled by the *OpenSync* build system and replaced with the value given in Kconfig option CONFIG_INSTALL_PREFIX. So all files placed in this directory in any of the rootfs section will be installed on the configured path.

By default CONFIG_INSTALL_PREFIX value is to “/usr/opensync”.

In the below example file “foo” will be installed on “/usr/opensync/foo”, in case Kconfig option CONFIG_INSTALL_PREFIX is set to “/usr/opensync”.

```
rootfs/
└── common
    └── INSTALL_PREFIX
        └── foo
```

Multiple OpenSync layers

Static files can be placed into different *OpenSync* layers, depending on which platforms they are needed and how generic is their need and/or implementation.

The rootfs/ directories in basic OpenSync layers:

- **core/rootfs/*** - generic static files
- **platform/*/rootfs/*** - platform specific static files
- **vendor/*/rootfs/*** - vendor or target specific static files
- **service-provider/*/rootfs/*** - service provider specific files

During build, more specific layers (ie: vendor/*/rootfs/target/*) may override layers which are more generic with default order defined with ROOTFS_SOURCE_DIRS build time variable.

In the below case file “/etc/foo” from “core” layer will be overwritten with “/etc/foo” from vendor “vndx” layer in case the given vendor is selected by the current build target.

```
core/rootfs/common/
└── etc
    └── foo
vendor/vndx/rootfs/common/
└── etc
    └── foo
```

Use of Jinja templating engine

Using the Jinja templating engine is also supported in files placed in rootfs/ directory. There are two options on how to tell the build system to preprocess files as Jinja templates:

1. Insert the following line in a static file: “# {# jinja-parse #}” (preferred option)
2. Use “.jinja” postfix for the name of a static file

OpenSync build system will process both types, following Jinja template syntax and using values supplied by current Kconfig settings.

In the below simple case file “rootfs/common/etc/foo” will be processed as a Jinja template and injected with Kconfig value.

```
$ cat rootfs/common/etc/foo
#!/bin/sh
# {# jinja-parse #}

ifconfig {{ CONFIG_TARGET_LAN_BRIDGE_NAME }} up
```

Installed file:

```
$ cat /etc/foo
#!/bin/sh

ifconfig br-home up
```

More documentation about Jinja templating engine is available here.

The ovsdb/ directory

The ovsdb directory allows you to add the pre-populated OVSDB entries at build time.
All the entries together are then built into a pre-populated ovs database necessary for initial operations.

The ovsdb/ directory is present in “core” and “vendor” opensync layers.

```
core
└── ovsdb
    ├── 00_openvswitch.json
    ├── ...
    ├── 20_kconfig.sta_backhaul_inet.json.sh
    ├── 40_node_services.json.jinja
    └── Kconfig
vendor
└── vndx
    └── ovsdb
        ├── common
        ├── TARGET_ABC
        └── TARGET_XYZ
```

“vendor” ovsdb/ directory is further divided into common and several targets.

Use of common/ovsdb/Kconfig file

This file is the only one in the ovsdb/ directory that is not an OVSDB entry.
Menuconfig part for setting OVSDB data, needed for extender onboarding, is implemented here. Lists of radio and wireless station interfaces as well as MTU size of GRE packet can be configured when 'make menuconfig' is executed from the core directory.

Use of ovsdb bootstrap scripts

Except the Kconfig, all other files in the ovsdb/ directory are pre-populated ovsdb entries, called also ovsdb bootstrap scripts. Together they are built in a pre-populated ovs database in json format.

As such, the scripts represent chunks of json ovs data. There are three types of files: json, shell and jinja (*.json, *.json.sh, *.json.jinja). While the json file is raw data already, with shell and jinja scripts you can add some logic as they generate raw data upon execution.

With current pre-populated ovs data, position of a json chunk, given by its entry name, in the generated ovs database is not important. As this might not hold for more complex cases, sorting chunks based on alphabetical order of their entry names is implemented. Therefore, it is recommended to start a script name with a two-digit number. The higher the number, closer to the end the position.

When building for a specific target, all ovsdb bootstrap scripts, i.e. *.json, *.json.sh, and *.json.jinja files from directories core/ovsdb/, vendor/vndx/ovsdb/common/, and vendor/vndx/ovsdb/TARGET_XYZ/ are processed.

After compilation you want to check if your pre-populated ovs database was generated as expected.

You can check for the file name in “core/ovsdb.mk” file, variable OVSDB_DB_NAME. Typically it is “conf.db.bck”. The content is a json string in only one very long line.

Processing the scripts

As can be seen in the core/build/ovsdb.mk file, an essential step in creating pre-populated ovs data from an ovsdb bootstrap script is:

```
$ ovsdb-tool transact ...
```

The command takes json from a script as an input, for example:

```
[
    "Open_vSwitch",
    {
        "op":"update",
        "table":"AWLAN_Node",
        "where":[],
        "row": { "led_config": ["map",[["state","connecting"]]] }
    }
]
```

As stated in the openvswitch ovsdb-tool documentation for transact: “Opens db, executes transaction on it, prints the results, and commits any changes to db. The transaction must be a JSON array….”.

More on ovsdb-tool http://www.openvswitch.org/support/dist-docs-2.5/ovsdb-tool.1.txt

Adding new vendors and targets to the build

This section gives an example for adding:

- a new vendor layer named "vndx"
- a new TARGET named "MODELX"

Complete the following steps:

1. Prepare a vendor layer directory layout based on the vendor templates for your SoC:
    a. Add *vendor/vndx* directory.
    b. Add directories below the *vendor/vndx*: build, rootfs, ovsdb, and src

Directory layout should look like this:

```
.
├── core
└── vendor
    └── vndx
        ├── build
        ├── ovsdb
        ├── rootfs
        └── src
```

2. Add makefile vendor/vndx/build/target-arch.mk:

```
# append list of all supported targets
OS_TARGETS += MODELX
ifeq ($(TARGET),MODELX)
VENDOR = vndx
ARCH = vndx
ARCH_MK = vendor/$(VENDOR)/build/$(ARCH).mk
# Set some build options (for all options see core/build/default.mk)
BUILD_LOG_PREFIX_PLUME                := n
BUILD_LOG_HOSTNAME                    := n
# Disable some unneeded units
UNIT_DISABLE_src/sample/unit/path     := y
endif
```

3. Add file *vendor/vndx/build/vndx.mk*:

```
# Set some compiling options:
INCLUDES     += -Isample/path
DEFINES      += -DSAMPLE_DEFINE
DEFINES      += -Wno-strict-aliasing
LDFLAGS      += -Lpath/to/additional/libs
LDFLAGS      += -ladditional_lib
SDK_ROOTFS   := path/to/sdk/rootfs
```

4. Add support for MODELX to the target library by creating *vendor/vndx/src/lib/target/override.mk*:

```
UNIT_CFLAGS += -I$(OVERRIDE_DIR)/inc
UNIT_EXPORT_CFLAGS := $(UNIT_CFLAGS)
UNIT_SRC_TOP += $(OVERRIDE_DIR)/src/target_MODELX.c
```

5. vendor/vndx/src/lib/target/inc/target_MODELX.h:

```
// MODELX - specific declarations
```

6. vendor/vndx/src/lib/target/src/target_MODELX.c:

```
// MODELX - specific target implementation
```

7. Additional files that need to be installed to rootfs (optional):

```
vendor/vndx/rootfs/common/etc/sample_file.txt
```

8. Define additional ovsdb database prepoluation rules (optional) (see chapter OVSDB and Cloud Schema):

```
vendor/vndx/ovsdb/sample.json.sh
```

Target Library

OpenSync standalone code provides core functionality that is integratable on a wide variety of devices. Device dependent code belongs in the target library and is different for each platform. Vendors will need to implement this library if a platform is not supported. Each device type is defined in *OpenSync* as its own “target”, the vendor must then supply the extra code that is specific for this target alone, e.g. initialization of the system, startup and shutdown scripts, management tools, Bluetooth manager, linking software between *OpenSync* and driver, etc.

Full target library documentation listing all APIs is provided in the *OpenSync 2.0 Southbound API* document [2].

Stubs

The target library provides a mechanism to automatically “stub” functions which are not implemented. This makes it easier to start porting *OpenSync* to a new target. Stubs allow you to selectively implement the functions that are needed for your specific device. Another benefit of stubs is that adding new APIs don't require simultaneous implementation and testing across all platforms/vendors, but instead, a default or empty implementation is provided. You can decide for individual targets to implement certain functionality or not.

Some stubs are no-ops that always return success, some stubs are in the form of a default implementation that can be used in the majority of cases, and some stubs only return dummy values. The stubs are used automatically if a function implementation is not provided within the target library. Implementation details can be found in:

`core/src/lib/target/unit.mk` and `core/src/lib/target/src/target_stub.c`

Target Library Interface

The following APIs are used by every manager to initialize the target subsystem:

- target_log_open
- target_init
- target_close

Managers

OpenSync consists of multiple managers running as separate processes and performing their specific set of tasks. Due to the functional dependencies, it is advised to do manager integration by the order given in the list below. Managers listed in the “Optional” group can be selectively left out. This depends on the desired functionality.

1. Required for basic *OpenSync* operation
    a. Diagnostics Manager
    b. Connection Manager
    c. WAN Orchestrator
2. Required for basic system network configuration
    a. Wireless Manager
    b. Network Manager
3. Optional, needed for additional features
    a. Queue Manager
    b. Statistics Manager
    c. Band steering Manager
    d. Log Manager
    e. Platform Manager
    f. Exchange Manager
    g. Upgrade Manager
4. OpenFlow management and configuration
    a. FlowService Manager
    b. OpenFlow Manager
    c. FlowCollection Manager

OpenSync is constantly developing. Use the table below to check the availability of managers in the existing releases:

Rel. no.	1.4	2.0	2.2
DM	✓	✓	✓
CM2	✓	✓	✓
CPM			✓
WANO			✓
WM2	✓	✓	✓
NM2	✓	✓	✓
QM	✓	✓	✓
SM	✓	✓	✓
BM	✓	✓	✓

LM	✓	✓	✓
PM	✓	✓	✓
XM	✓	✓	✓
UM		✓	✓
FSM	✓	✓	✓
OM	✓	✓	✓
FCM	✓	✓	✓

Note: *The procedure for adding new managers is available in document EIN-020-032-501 Adding new managers to OpenSync 2.0 [3].*

Diagnostics Manager - DM

Diagnostics manager is the first manager that needs to get started. It is responsible for spawning the rest of the *OpenSync* managers and optionally monitoring them.

Startup

The diagnostics manager can be started in several ways, like with *systemd* or through */etc/init.d* scripts, an example of the latter is shown here:

```
start () {
    # OpenSync start with service-start
    service_start ${BIN_DIR}/dm
}

stop() {
    killall -s SIGKILL dm cm nm wm lm sm qm
}
```

Note that *ovsdb-server* needs to be started prior to DM.

```
ovs_start () {
    # ovsdb-server start against non-persistent DB with service-start
    service_start ${OVS_BI_DIR}/ovsdb-server \
        --remote=punix:${OVS_RUN_DIR}/db.sock \
        --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
        --private-key=db:Open_vSwitch,SSL,private_key \
        --certificate=db:Open_vSwitch,SSL,certificate \
        --ca-cert=db:Open_vSwitch,SSL,ca_cert \
        --pidfile=${OVS_RUN_DIR}/ovsdb-server.pid \
        ${OVS_ETC_DIR}/conf.d
}

ovs_stop() {
    # ovsdb-server stop with service stop
    service_stop  /usr/sbin/ovsdb-server
}
```

When/If ovs-vswitchd is used it needs to be started after ovsdb-server and prior to DM!

Initialization

Diagnostic manager progresses through different states of initialization to assure that the system is ready before starting the managers. Requirements that need to be met:

1. Check system readiness - specified by **target layer**
2. Check OVSDB server connection
3. Read entity data (device information)
4. Execute hooks (optional)
5. Start managers

OVSDB Interface

Beside checking system readiness the DM is responsible to fill mandatory device entity information needed for the Cloud to know how specific devices need to be managed. Not all information is mandatory, but Cloud determines functionality supported by the devices based on this information, so it is recommended to implement all of the functions.

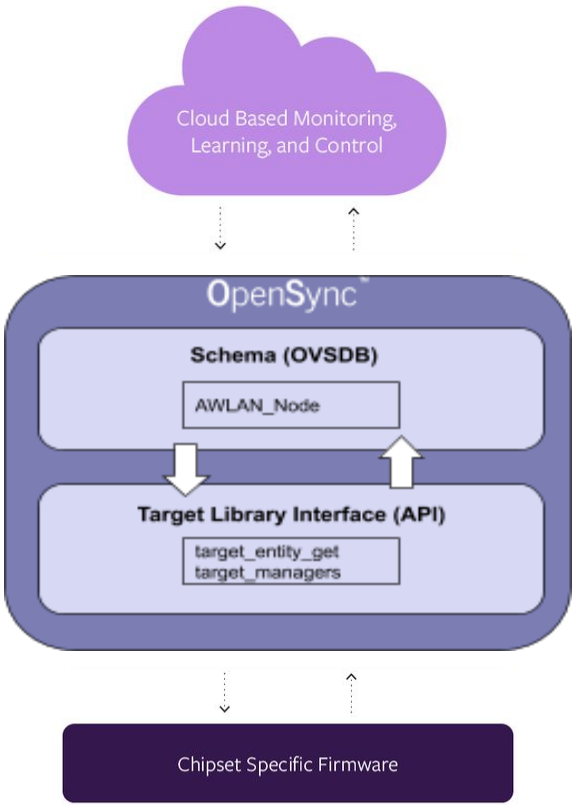


Figure 3: *OpenSync Diagnostic Manager block diagram*

Managed OVSDB tables:

- AWLAN_Node
- *Node_Services*

The most important settings for the *AWLAN_Node* table are:

- id
- model
- serial_number
- firmware_version
- redirector_addr

The most important settings for the *Node_Services* table are:

- service
- enable
- other_config (more in Manager Configuration)

More details regarding *AWLAN_Node* and *Node_Services* tables can be found in the *OpenSync 2.2 Northbound API* document [1].

Target Library Interface

- target_ready
- target_serial_get
- target_id_get
- target_ethclient_brlist_get
- target_ethclient_iflist_get
- target_sku_get
- target_model_get
- target_sw_version_get
- target_hw_revision_get
- target_platform_version_get
- target_managers_restart
- target_managers_config

Monitoring

Beside target layer APIs, responsible for initializing the abstraction interface layer, diagnostics manager provides two additional hooks that can be used for initializing DM features at initialization time and at manager shutdown. These are placeholders for vendor implementation, for which *OpenSync* only returns true.

```
bool dm_hook_init(struct ev_loop *loop) {
    (void)loop;
    return true;
}

bool dm_hook_close() {
    return true;
}
```

Connection Manager - CM(2)

CM2 is responsible for establishing the backhaul connection and keeping connectivity to the Cloud via IPv4 or IPv6. Connectivity is realized using the following steps:

1. Uplink selection (method depends on device type)
2. Get IP address based on hostname from **redirector_addr** in *AWLAN_Node* OVS table
3. Set up a connection to the redirector
4. Get an IP address for the specified cloud by the redirector
5. Connect manager to the Cloud
6. Monitor link stability, on disconnect repeat process

The uplink selection depends on device type:

- Gateway - for gateway devices the uplink is always a cabled port (ethernet, fiber, coax, etc.). No special logic is applied in this case, only connection to the controller is monitored.
- Extender - in this case, CM will select the appropriate uplink port - either a cabled port or wifi backhaul.

OVSDB Interface

Connection manager uses *AWLAN_Node* **redirector_addr** to get **manager_addr** required to establish a connection to a specified Cloud. The manager address depends on the Cloud to which the pod is claimed.

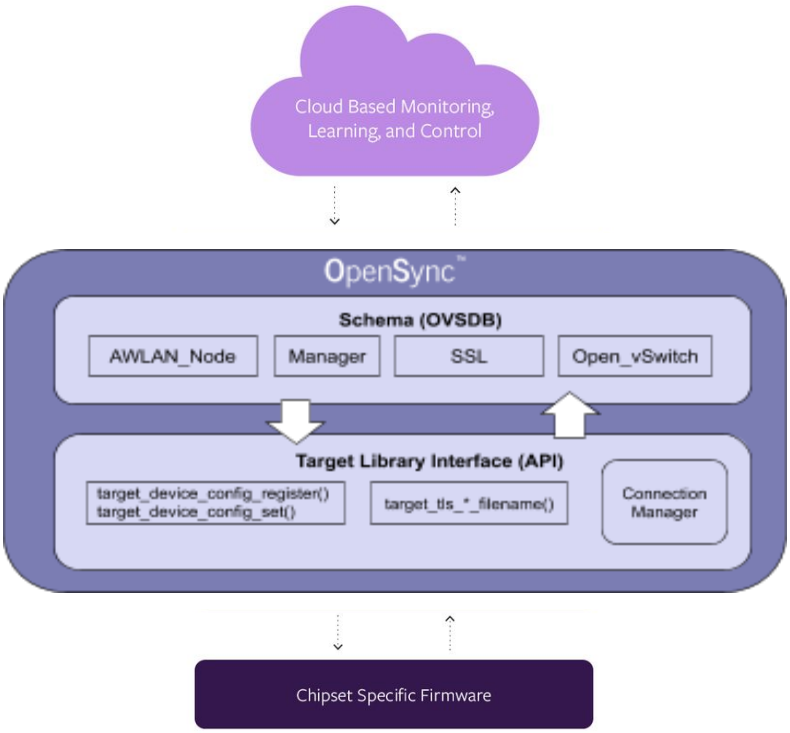


Figure 4: *OpenSync Connection Manager block diagram*

Managed OVSDB tables:

- *Open_vSwitch*
- *Manager*
- *SSL*
- *AWLAN_Node*
- *Connection_Manager_Uplink*
- *Wifi_Master_State*
- *AW_Bluetooth_Config*
- *Wifi_Inet_Config*
- *Wifi_Inet_State*

- *Wifi_VIF_Config*
- *Wifi_VIF_State*
- *Port*

The *Open_vSwitch, Manager* and *SSL* tables are used to keep the status of Cloud connection and list information about current connectivity status.

AWLAN_Node contains min_backoff and max_backoff values to protect the Controller against overloading by triggering multiple connections to the controller at the same time. Backoff time is calculated based on random value between min_backoff and max_backoff.

When the device operates as Extender the *AW_Bluetooth_Config* table shows onboarding state presented by payload value.

For tracking of uplink port selection and status, the following OVSDB tables are used:

- *Wifi_Master_State*
- *Wifi_Inet_Config*
- *Wifi_Inet_State*
- *Wifi_VIF_Config*
- *Wifi_VIF_State*
- *Port*

More details regarding the above tables can be found in *OpenSync 2.0 Northbound API* document [1].

Target Library Interface

The following target library interfaces are used by CM2:

- target_device_config_register - get platform-specific redirector address from target storage
- target_device_config_set - set configuration in the target based on AWLAN_Node table
- TLS properties:
    - target_tls_cacert_filename
    - target_tls_mycert_filename
    - target_tls_privkey_filename
- target_bin_dir
- target_device_capabilities_get
- target_device_connectivity_check
- target_device_execute
- target_device_restart_managers
- target_device_wdt_ping

Captive Portal Manager - CPM

CPM runs a proxy service on the *OpenSync* devices that enables authentication of third party (guest) clients in private Wi-Fi networks. The guest client devices are authenticated via external Captive Portal servers.

The CPM has the following roles:

- Identifying the traffic that needs to get authenticated
- Routing the traffic to the proxy service
- Passing the proxy service from the received HTTP port 80 traffic to the UAM/NAS service running in the Cloud

OVSDB Interface

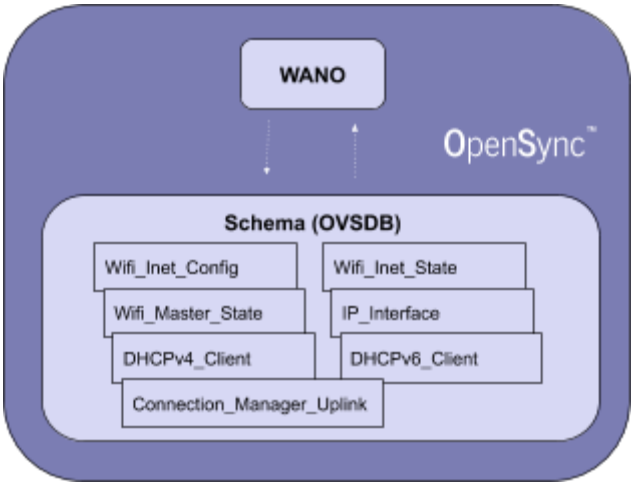
- Captive_Portal

WAN Orchestrator

WAN Orchestrator (WANO) enables internet connectivity for the onboarding devices. WANO brings up the device WAN interface and activates the necessary services to enable internet connectivity.

These are the steps when provisioning the WAN connection:

1. Interface creation. WANO supports these interface types: Ethernet, VLAN, and PPPoE.
2. Port role detection and WAN probing:
    a. WANO automatically configures the Ethernet ports, and enables autodetection for these port roles: WAN provisioning, Ethernet backhaul, Ethernet extension.
    b. Activating security and enabling firewall rules before WAN provisioning.
    c. WANO probes for the available WAN interfaces:
        - Without a persistent WAN configuration on the device, WANO defaults to DHCP probing.
        - With a static IP configuration loaded, WANO probes the well-known internet hosts.
        - WAN config for PPPoE, VLAN.
3. Network provisioning. The WANO is in charge of WAN provisioning on the devices.



OVSDB Interface

- Wifi_Inet_Config
- Wifi_Inet_State
- Wifi_Master_State
- IP_Interface
- DHCPv4_Client
- DHCPv6_Client
- Connection_Manager_Uplink

Wireless Manager - WM(2)

Initially WM reads the current configuration of the device and updates the Config and State tables. The Cloud uses a number of radios and interfaces with a model name to properly set Adaptive WiFi parameters.

WM is responsible for:

- Maintaining *Wifi_Radio_Config*, *Wifi_Radio_State*, *Wifi_VIF_Config* and *Wifi_VIF_State* in sync, ie. if the corresponding State row has columns which don't match intended Config, If Config row's column is *["set", []]* then the State column can be whatever and it will be considered matching. WM2 will call target API implementation to configure upon mismatch.
- Reporting connected clients and their metadata via *Wifi_Associated_Clients*
- Updates Wifi_Master_State *port_active* column whenever STA VIF link state changes in order to notify CM. This is used for onboarding and uplink connectivity on Extenders.

OVSDB Interface

WM performs Radio and VIF row recalculations as deferred jobs. These jobs are queued up upon Config and State table updates. These updates can both come from target API implementation (both Config and State) and from OVSDB (Config only).

WM hands over a set of callbacks to target API implementation over *target_radio_init* by giving out *target_radio_ops*. These allow updating Config and State rows for Radio and VIF tables. They also allow managing connected Clients.

Configuration tables are written by the Cloud and locally by WM. The latter is used exclusively for GW devices. Extenders do not modify Config tables.

Status tables are updated by WM and are read by the Cloud or other managers, eg. SM. Status tables do not only reveal the radio and interface status but also provide information about connected clients and their fingerprint.

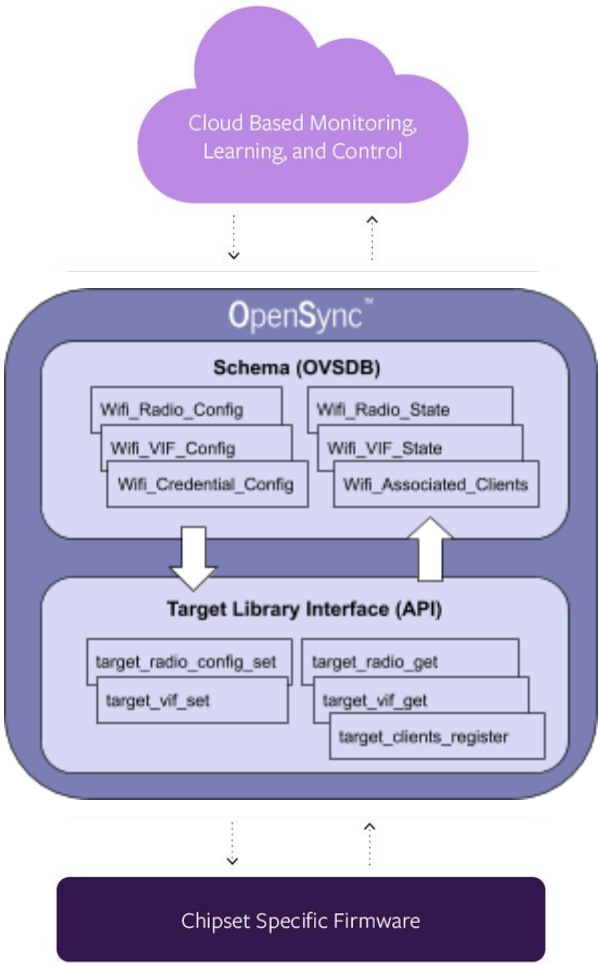


Figure 5: *OpenSync Wireless Manager block diagram*

Managed OVSDB tables:

- *Wifi_Radio_Config*
- *Wifi_VIF_Config*
- *Wifi_Radio_State*
- *Wifi_VIF_State*
- *Wifi_Associated_Clients*
- *Wifi_Master_State (partial)*

The most important settings for *Wifi_Radio_Config* table are:

- beacon interval
- channel
- channel_mode
- country
- ht_mode

- interface name
- tx_chainmask
- tx_power

The most important settings for *Wifi_VIF_Config* table are:

- mode (ap/sta)
- interface name
- security (PSK/FT/...)
- bridge
- ssid
- ssid broadcast
- mac white-list (important for backhaul network)
- group rekey
- Uapsd

More details regarding the above tables can be found in *OpenSync 2.2 Northbound API* document [1].

Target Library Interface

Target implementation calls[2]

- *op_vconf* - whenever the upper management layer changes VIF config locally. This is applicable mostly for GW systems. GWs are expected to call this in *target_radio_config_init2* implementation. Extenders do not use this because they are managed exclusively by the controller.
- *op_vstate* - whenever wireless driver VIF state changes, e.g. channel is switched, ssid reconfiguration completes, etc.
- *op_rconf* - same as *op_vconf*, but for Radio
- *op_rstate* - same as *op_vstate*, but for Radio
- *op_client* - whenever a Client connects or disconnects, or it's *key_id* becomes known
- *op_clients* - same as *op_client*, but this always provides a complete Client list. If a Client with some mac address was reported once, and then subsequent *op_clients* calls does not contain it, the Client is considered disconnected. This is more robust, but depending on the target API internals, may be computationally more expensive to implement.
- *op_flush_clients* - whenever target API implementation suffers internal event loss, or errors, it is encouraged to call this function and then re-report clients with *op_client*, or *op_clients*.

Target implementation provides

- *target_radio_init* - this is used for WM to advertise *target_radio_ops* to target (see above)

[2] See target_radio_init in the subsequent listing.

- *target_radio_config_need_reset* - if target wants to start from scratch upon WM start up, it needs to return true. This is intended for GW systems where they need to interact with a local configuration authority. Extenders are managed solely by the cloud controller, so they return false.
- *target_radio_config_init2* - this is called by WM only if *target_radio_config_need_reset* is true, and is intended to allow target to bootstrap Radio and VIF tables, both Config and State.
- *target_radio_config_set2* - this is called by WM when it is concluded that Radio Config and State rows have any of their columns mismatched. It hands over a hint which columns have changed, along with a complete Radio config to aid things like hostapd config regeneration.
- *target_vif_config_set2* - this is called by WM when it is concluded that VIF Config and State rows have any of their columns mismatched. It hands over a hint which columns have changed, along with a complete VIF config to aid things like hostapd config regeneration.

Network Manager - NM(2)

The network manager is responsible for managing all network related configuration and network status reporting. Among others, its primary role is to:

- Manage IPv4/IPv6 addresses (static, DHCP)
- Create and destroy network interfaces (with the exception of Wifi interfaces, see WM)
- Configure interface parameters (MTU, Up/Down State)
- Manage DNS and DNSMASQ services
- Manage firewall rules (filtering, port forwarding)
- Start/stop various networking services (UPnP, DHCP server and clients)
- Manage GRE tunnels
- Manage bridge interfaces
- Associated Client DHCP fingerprint reporting
- Manage DHCP reserved IPs

The NM can manage OVS bridge, native bridge (functionalities like HomePass™ are missing), network interfaces and their configuration and Wifi interfaces (Access Point and Station).

The Cloud schema is an extension to the default OVSDB schema, which already provides all the necessary means for configuring OVS bridges. Wifi interfaces cannot be fully configured without interaction with the Wifi layer. For this reason, the creation semantics are more aligned with the functionality provided by WM.

OVSDB Interface

The network manager interacts with several OVSDB tables, all of which are Cloud specific extensions to the basic OVS schema. NM understands two general types of tables; configuration and status tables.

Configuration tables are typically written by the Cloud or other managers (for example, CM) and are read by NM. NM uses these tables to apply network configuration to the system.
Status tables are written/updated by NM and are read by the Cloud or other managers (CM, WM). These tables are used for reporting the current active network configuration and network status (link status, DHCP leases, ARP tables, etc.).

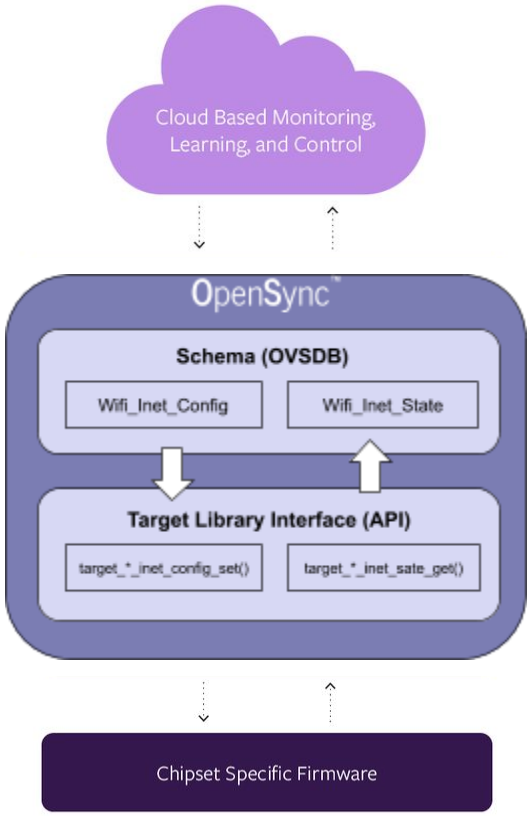


Figure 6: *OpenSync Network Manager block diagram*

Managed OVSDB tables:

- *Wifi_Inet_Config*
- *Wifi_Inet_State*
- *Wifi_Master_State*
- *OVS_MAC_Learning*
- *DHCP_leased_IP*
- *DHCP_reserved_IP*

Target Library Interface

The target library interface is used to realize the platform-specific network configuration and status reporting. The functions described below must be implemented when porting to a new platform in order to provide the required NM functionality.

```
struct schema_Wifi_Inet_Config;
```

A structure of type *schema_Wifi_Inet_Config* represents a single row in *Wifi_Inet_Config*; this is an auto-generated structure derived from the OVS schema and is typically read by NM from OVSDB. It is used as an input parameter to configuration functions.

```
struct schema_Wifi_Inet_State;
```

A structure of type schema_Wifi_Inet_State represents a single row in *Wifi_Inet_State*; this is an auto-generated structure derived from the OVS schema and is typically written by NM to OVSDB. It is used as an output parameter to state acquiring functions.

Net Filter Manager - NFM

Net Filter Manager - NFM will be used in upcoming releases for managing firewall specifics functions. It is included in this release but it is not activated. The Firewall functions are still managed by Network Manager - NM(2)

Stats Manager - SM

The Stats Manager is responsible for processing all requested wireless statistics and sending results to the Cloud. The configuration is done through OVSDB while MQTT is used for the data plane.

It is best to illustrate the process of collecting and sending stats with an example for client frequency (not a real stat measurement, only illustrative):

Clouds set a timer for collecting client frequency information every 10s and another timer for sending client reports every 120s. On timeout, a client frequency measurement is requested from the target layer. **Target layer** must provide an implementation of measurement which is usually device specific, like a kernel driver call. The target layer is also responsible for translating data structures it measures to the format that SM expects - this is defined in the data-pipeline - DPP library in *OpenSync*. Individual measurements are stored into a report, aggregated with other measurements and data and possibly processed and manipulated further and on report timer timeout, the report is sent to the Queue Manager which aggregates reports from different managers into a single one and sends it to the Cloud via MQTT.

OVSDB Interface

The OVSDB schema *Wifi_Stats_Config* defines sampling and reporting statistical behavior.

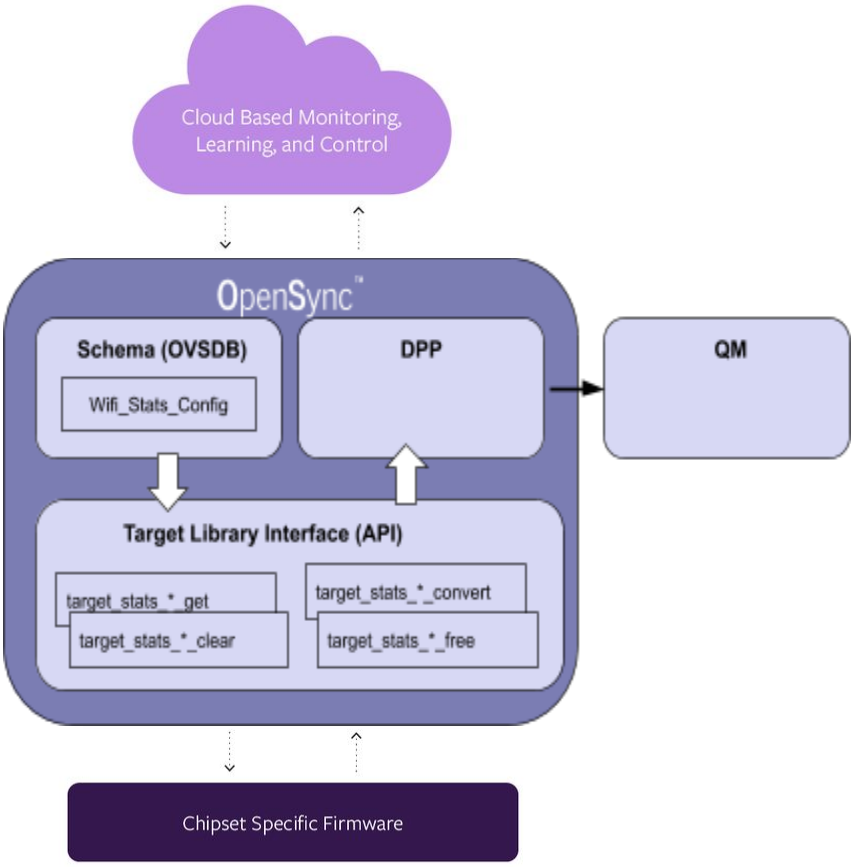


Figure 7: *OpenSync Statistic Manager block diagram*

Managed OVSDB tables:

- Wifi_Stats_Config

Target Library Interface

Stats are collected periodically, where at SM caches deltas between initial values (collected at startup) and following ones **target_get**

Please note that the ***target_convert*** function needs to convert different data to deltas depending on how the driver stores them:

- **consecutive:** most of the drivers maintain cached values of statistical information. In this case, *target_convert* will send **previous** and **new values** where **delta is = new - prev**
- **clear on get** some drivers clear data upon retrieval to minimize caching size. In this case **delta = new**

APIs used:

- target_client_record_alloc
- target_client_record_free
- target_is_interface_ready
- target_is_radio_interface_ready
- target_radio_fast_scan_enable
- target_radio_tx_stats_enable
- target_stats_capacity_convert
- target_stats_capacity_enable
- target_stats_capacity_get
- target_stats_clients_convert
- target_stats_clients_get
- target_stats_device_get
- target_stats_device_temp_get
- target_stats_scan_get
- target_stats_scan_start
- target_stats_scan_stop
- target_stats_survey_convert
- target_stats_survey_get
- target_survey_record_alloc
- target_survey_record_free
- target_stats_device_fanrpm_get
- target_stats_device_txchainmask_get

The SM target layer allows the user to specify target structure where some of the rules APIs need to be provided for cleanup. All APIs return value in the target linked list where each vendor can specify target structures sent to SM. The SM caches these structures and sends them through *target_convert* API which defines storage type.

Example
The tx_packet can be stored inside the driver as uin32 or uint64, and because the SM uses deltas this needs to be **converted** properly inside the *target_convert* API.

```
#if defined (TARGET1)
typedef target_1_client_t target_client_record_t;
#else
typedef target_2_client_t target_client_record_t
#endif
```

Using the linked list approach requires some additional APIs for cleaning (**target_free** and **target_alloc**) and structure rules.

Example: target_client.h

```
#if defined (TARGET1)
typedef struct
{
    /* Client general data */
    dpp_client_info_t               info;

    /* Target specific client data */
    uint32_t                        channel
    ds_dlist_node_t                 node;
} target_1_client_t;
#else
typedef struct
{
    /* Client general data */
    dpp_client_info_t               info;

    /* Target specific client data */
    uint64_t                        channel
    ds_dlist_node_t                 node;
} target_2_client_t;
#endif

static inline
target_1_client_t *target_1_client_record_alloc()
{
    return calloc(1, sizeof(target_1_client_record_t));
}

static inline
void target_1_client_record_free(target_1_client_t *record)
{
    if (NULL != record)  free(record);
}
```

Note: The same logic applies to all stats.

Data Pipeline Library

Depending on *Wifi_Stats_Config,* the Cloud can configure fetching of the following statistic, where some are available by default and some need to be extended. All statistics are sent through MQTT using protobuf schema (*interfaces/plume_stats.proto*). Internally all **target** samples need to be converted to DPP structures defined in the data pipeline library.

```
src/lib/datapipeline
```

DPP managed structures:

- dpp_neighbor
- dpp_survey
    - Off-chan (full) survey
    - On-chan survey
- dpp_client

    - Client average statistics
    - RX/TX client statistics
    - TID sojourn statistics
- Dpp_device
    - Thermal, fan rpm
    - Memory/CPU utilization
- dpp_rssi

Band Steering Manager - BM

Responsible for steering of wifi clients to a different band (band steering) or a different AP (client steering) based on per-client configuration, signal/noise strength and triggers provided by the Cloud.

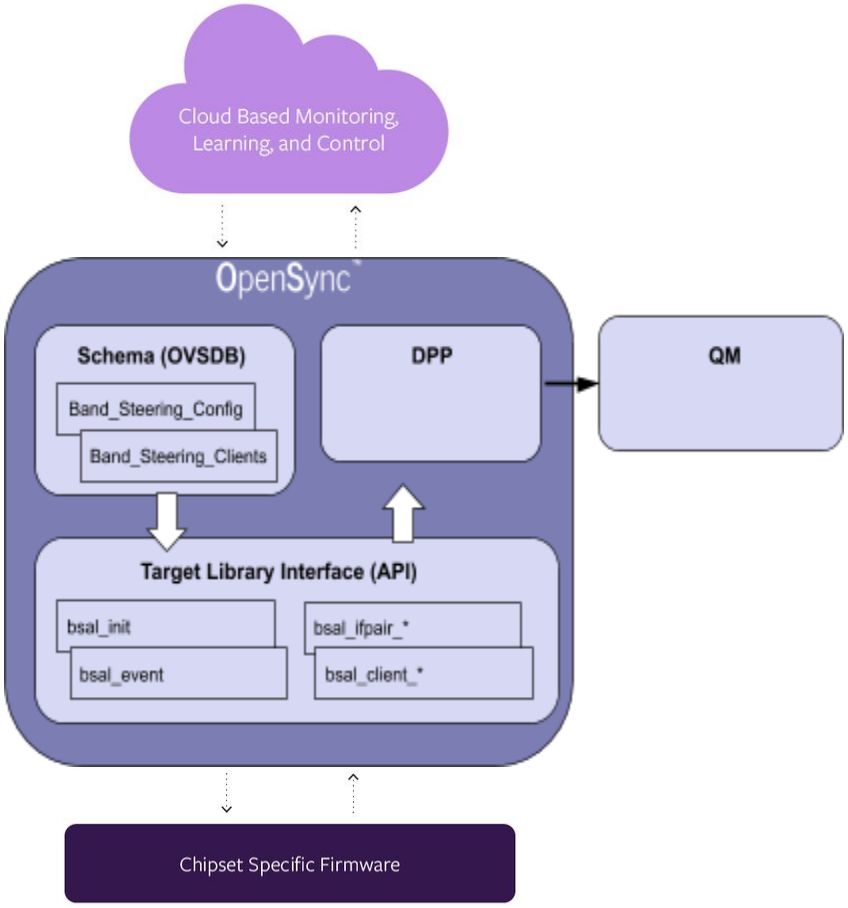


Figure 8: *OpenSync Band Steering Manager block diagram*

OVSDB Interface

The OVSDB tables are used for both Band and Client Steering configuration. The special target layer implementation BSAL - band steering abstraction layer - is used to configure the pre and post association client behavior.

The events collected during the steering are sent to the Cloud through Queue Manager which aggregates reports from different managers (like SM) into a single one and sends it to the Cloud via MQTT.

Managed OVSDB tables:

- Band_Steering_Config
- Band_Steering_Clients

Target library Interface

- target_bsal_init
- target_bsal_cleanup
- target_bsal_iface_add
- target_bsal_iface_update
- target_bsal_iface_remove
- target_bsal_client_add
- target_bsal_client_update
- target_bsal_client_remove
- target_bsal_client_measure
- target_bsal_client_disconnect
- target_bsal_client_info
- target_bsal_bss_tm_request
- target_bsal_rrm_beacon_report_request
- target_bsal_rrm_set_neighbor
- target_bsal_rrm_remove_neighbor

Data Pipeline Library

The band steering statistics can be divided into two segments

- **Operational:** Statistics about general steering behavior, successful steering, errors, etc.
- **Events:** Events in time manner containing information about connectivity status and probes

DPP managed structures:

- dpp_bs_client_record_t
    - Includes client capabilities (max mcs, max nss, 2ghz/5ghz, etc.)
    - Includes steering counters (attempts, failures, probe request counts, etc.)
    - Includes rssi, last disconnection reason, etc.

Queue Manager - QM

Responsible for buffering MQTT messages from SM and BM. Tries not to drop any messages even if uplink connectivity drops (e.g., transient failure of backhaul connectivity, ISP blackout, etc). The messages are encoded using Google protobuffers. Multiple messages within a reporting interval are merged into a single message. Other managers communicate with QM over a Unix socket.

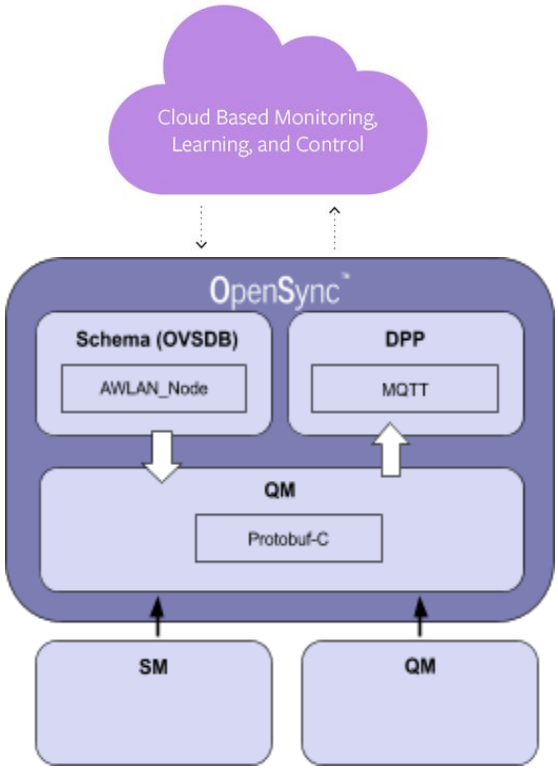


Figure 9: *OpenSync Queue Manager block diagram*

OVSDB Interface

QM uses the *AWLAN_Node* table, *mqtt_settings* field, to get all the required parameters (MQTT broker fqdn, port, compression, QoS, and topics) to establish MQTT connection.

Managed OVSDB tables:

- AWLAN_Node

Target library Interface

- None

Log Manager - LM

Log Manager is responsible for collecting and uploading logs and system information upon the Cloud request (logpull) and for handling the log severity setting for running modules. Collected data is the main source of information during on-site problem investigation.

Note: *With the Kconfig options for PM, you can define either to build a separate LM or as a part of PM.*

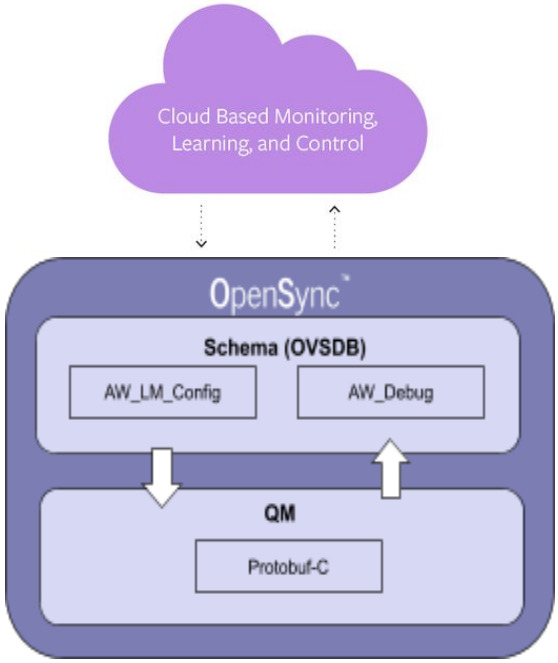


Figure 10: *OpenSync Log Manager block diagram*

OVSDB Interface

Managed OVSDB tables:

- *AW_LM_Config*
- *AW_LM_State - deprecated*
- *AW_Debug*

Target Library Interface

- target_log_pull - intended to call custom method for collection of device logs and other information in order to upload to the cloud.
- target_log_state_file
- target_log_trigger_dir
- target_log_open

OpenFlow Manager - OM

In case OpenVSwitch is used on the device the OM is responsible for managing packet flow rules. Packet flow rules provide features such as client freeze or HomePass™. OpenVSwitch must be operational to use OpenFlow manager.

OM is responsible for:

- Applying flow directly from the controller
- Applying dynamic flows based on tags given by the controller

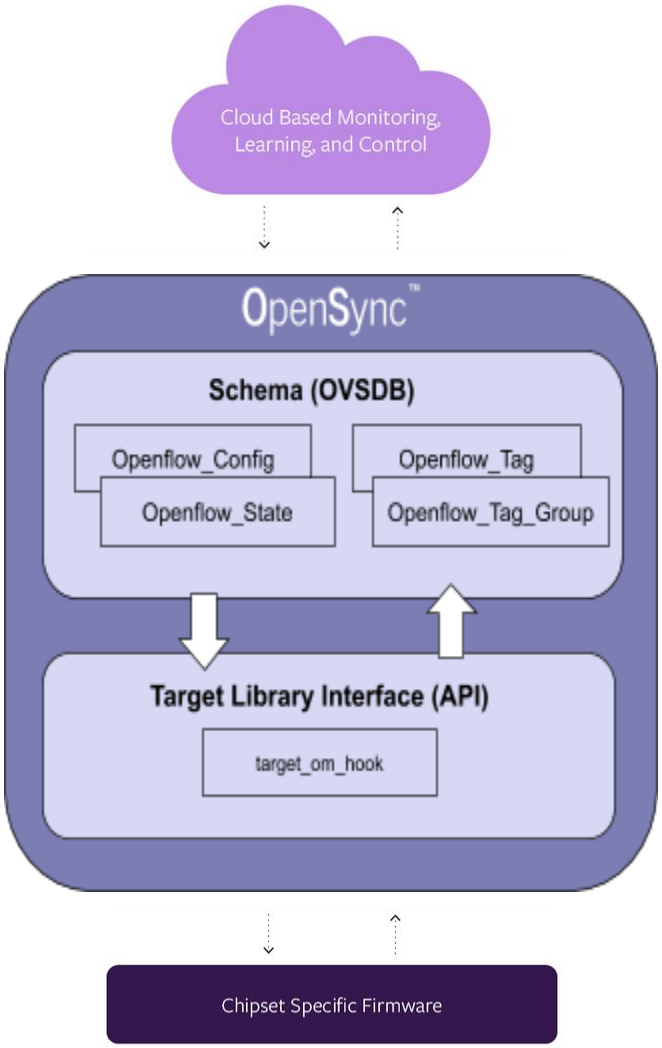


Figure 11: *OpenSync OpenFlow Manager block diagram*

OVSDB Interface

Managed OVSDB tables:

- *Openflow_Config*
- *Openflow_State*
- *Openflow_Tag*
- *Openflow_Tag_Group*

Openflow_Config

This table contains all the flows that need to be applied to the system.

Name	Type	Description
action	string	Flow action (drop, normal, submit, etc.)
bridge	string	Bridge name
priority	integer	Flow priority
rule	string	Flow rule or flow rule template
table	integer	Flow table
token	string	Name of the flow

Openflow_State

This table reflects the status of all currently applied flows.

Name	Type	Description
bridge	string	Bridge name
openflow_config	string	NOT USED
success	bool	Marks successfully applied flow
token	string	Name of the flow

Openflow_Tag

This table is used for packet flow rules expansion.

Name	Type	Description
cloud_value	string	Cloud only tag value
device_value	string	Device only tag value
name	string	Tag value name

Openflow_Tag_Group

This table is used for packet flow rules expansion.

Name	Type	Description
tags	string	Reference to tag in Openflow_tag table
name	string	Tag group name

Target Library Interface

One function is used to perform actions before or after manipulating packet flow rules:

- target_om_hook

Flow Service Manager - FSM

FSM provides infrastructure for managing the data traffic flows.

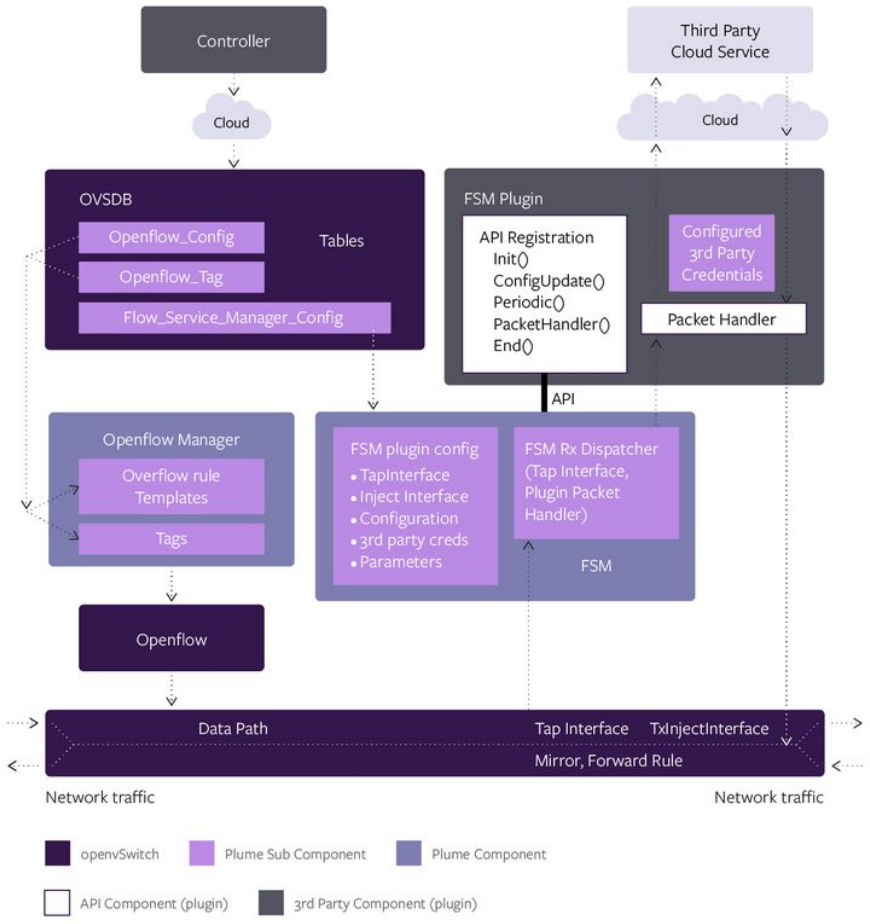


Figure 12: *OpenSync Flow Service Manager block diagram*

The diagram shows the flow monitoring and control portions of *OpenSync*. These portions of *OpenSync* provide the foundation of the access control service. In addition, they provide the required hooks for implementing cyber security, parental controls, and future services.

The key to operation of FSM is the ability of OVS to monitor and manipulate the relevant traffic flows. OVS is used to block, filter, forward, and prioritize the messages. All of these OVS activities are programmable from the Cloud using OVSDB.

OVSDB Interface

Managed OVSDB tables:

- *Openflow_Tag*
- *Openflow_Tag_Group*
- *AWLAN_Node*
- *Flow_Service_Manager_Config*
- *FSM_Policy*
- *DHCP_leased_IP*
- *IPv6_Neighbors*

Target Library Interface

- None

FSM ships with various plugins. These plugins enable provisioning of various third party cloud-based services, such as

- SpeedTest
- AI Security™
    - Online protection
    - IOT security
- Adblocking
- Content filtering
- Digital Wellbeing
- Physical Wellbeing
- Application blocking

Read more about providing the *OpenSync* services in document EUB-020-060-301 OpenSync Service Portfolio

Flow Collection Manager - FCM

Flow Collection Manager (FCM) is responsible for collection and reporting of network traffic statistics, occurring between various end-clients in the LAN and WLAN networks, and also between clients in (W)LAN network and devices in WAN network. The main functionalities are

collecting the network statistics at regular intervals, storing the collected statistics of flows in a cache, and reporting the network statistics to the cloud in protobuf encoded format via MQTT.

FCM provides advanced functionalities for collecting and reporting the network statistics using filters. The FCM filters rules will be applied to the network flows. The network flows are then included or excluded based on the configured filter criteria. Filters can be applied at collection time as well as at reporting time.

FCM is capable of supporting multiple plugins, such that different types of network stats collection/reporting methods can be separately managed by the Cloud. Currently, FCM support these plugins:

1. Lan-to-Lan Stats Plugin - Used to collect and report layer 2 network statistics occurring between devices within the LAN network.
2. IP Flow Stats Plugin - Used to collect and report IP flow statistics occurring between devices in the LAN network to devices in the WAN network and vice-versa.
3. Per-interface Bandwidth Stats Plugin - Tracks the total bandwidth consumed on all wired and wireless interfaces. The collected data:
    - serves as an accurate count of the WAN data consumption
    - helps monitor the WAN-side data saturation
    - provides load indication for each network within the SSID

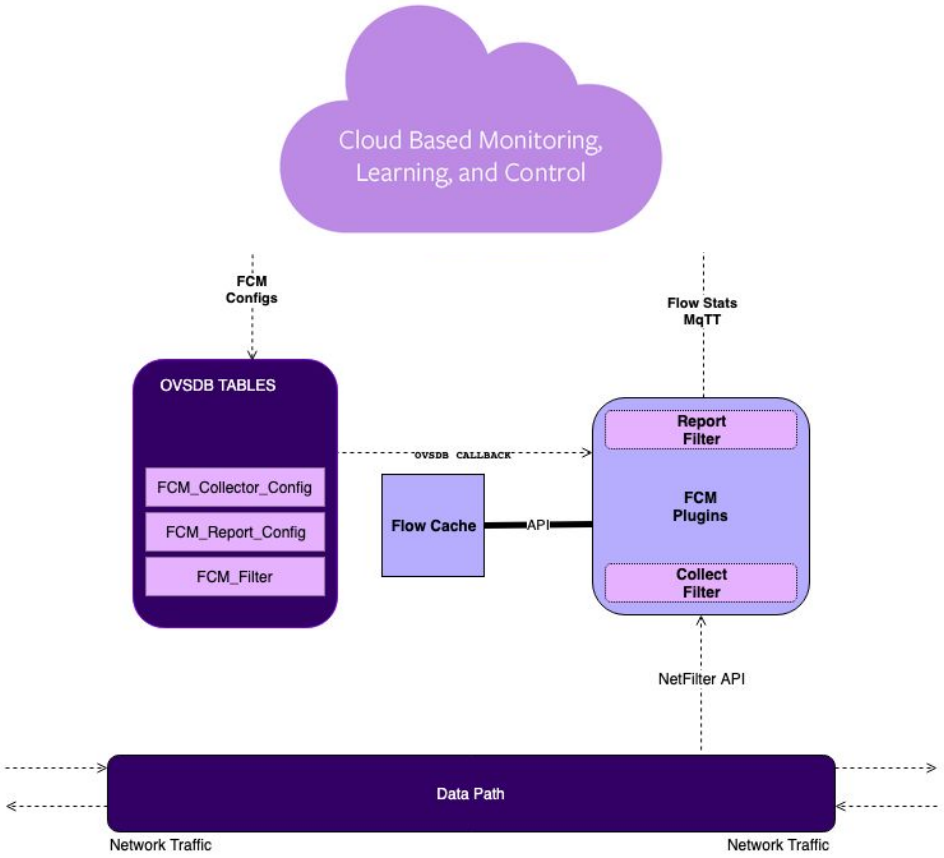


Figure 13: *OpenSync Flow Collection Manager block diagram*

OVSDB Interface

Managed OVSDB tables:

- FCM_Collector_Config
- FCM_Report_Config
- FCM_Filter
- DHCP_leased_IP
- IPv6_Neighbors

Target Library Interface

- None

Platform Manager - PM

The purpose of the Platform Manager (PM) is to cover specific platform-related features which can not be covered by other managers. With *OpenSync* release 2.0, PM has additionally become responsible for certain features that were previously covered by other managers.

Currently, the PM is responsible for:

- Nickname synchronization between device GUI and the Cloud
- Cloud-managed device parental control aka device freeze[3]
- Device thermal management by regulating fan rotation speed
- Device LED management
- Log management

[3] OpenFlow must be disabled

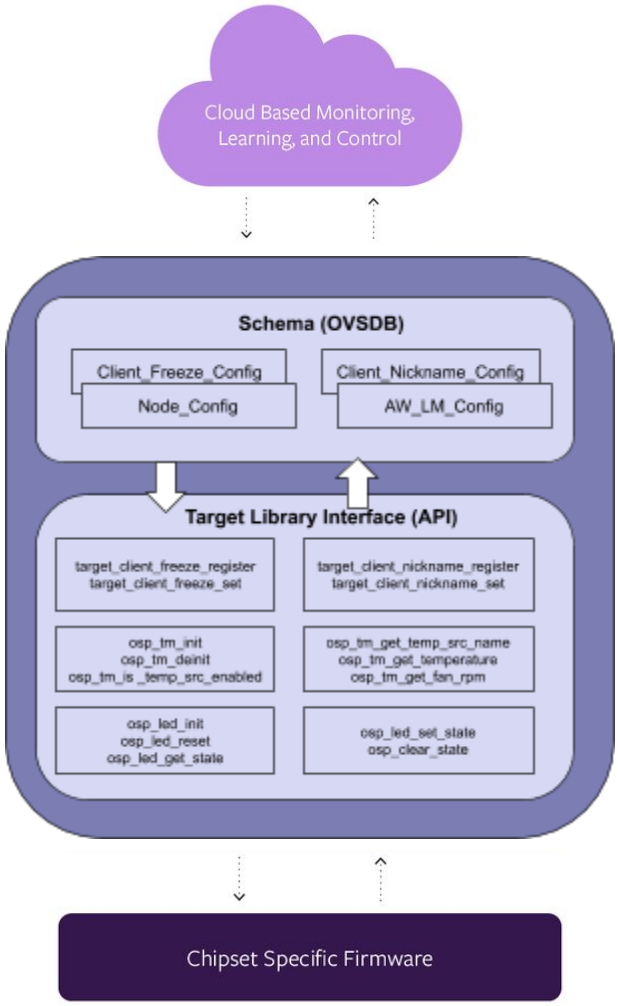


Figure 14: *OpenSync Platform Manager block diagram*

To enable a specific feature, use the below listed flags:

```
PM_ENABLE_CLIENT_FREEZE          :=y
PM_ENABLE_CLIENT_NICKNAME        :=y
PM_ENABLE_LED                    :=y
PM_ENABLE_TM                     :=y
PM_ENABLE_LM                     :=y
```

By default, these flags are all disabled. To enable the flags, use the Kconfig tool. After you open the Kconfig, go to **Common > Managers > Platform Manager (PM).** Use the available options to enable the individual features (flags):

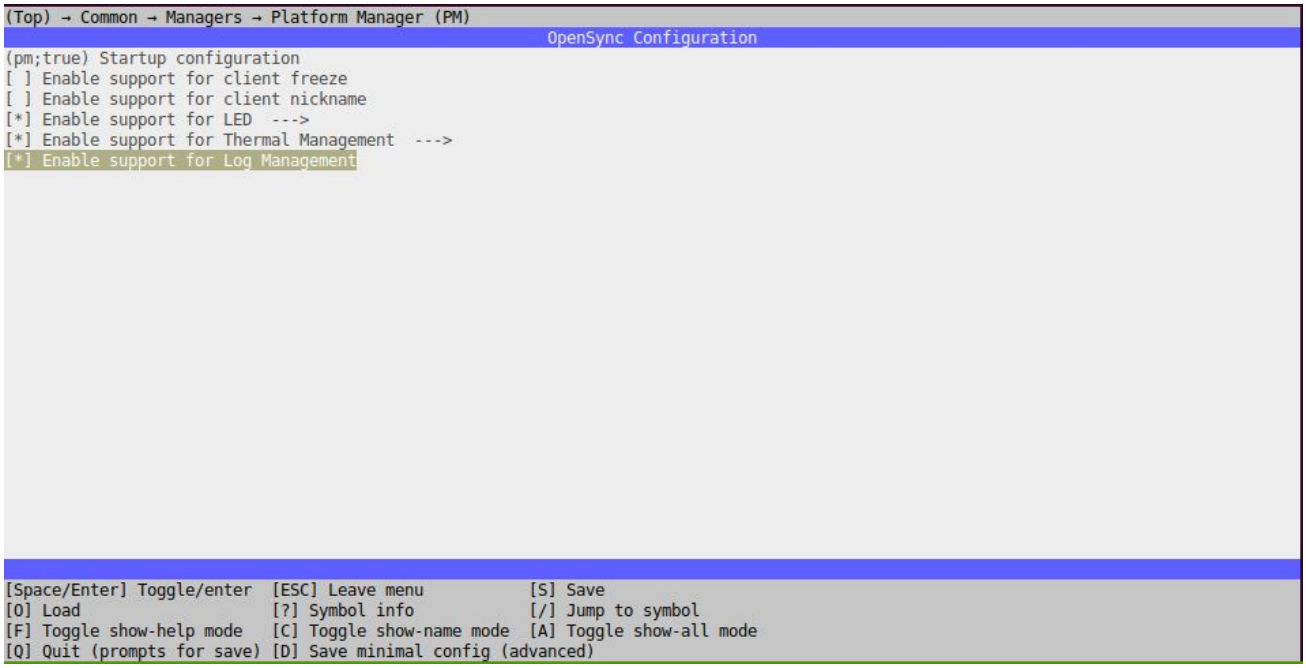


Figure 15: *Configure OpenSync Platform Manager with Kconfig*

With the Kconfig options for PM (src/pm/kconfig/Kconfig.managers), you can define either to build separate managers (LED, log, thermal) or to build their functionality as a part of PM.

Maintaining Local Connectivity in Case of Internet Outage

In cases of lost connection to the internet or Cloud, the devices at a location keep their LAN side connectivity. Even after reboot, the devices at a disconnected location keep their configuration, including:

- Running in bridge or router mode
- Keeping the existing static DHCP reservations and user-configured subnet
- Keeping the user-configured primary/secondary DNS servers
- Keeping home SSID & primary PSK (single zone, no guest/internet only)

After rebooting the gateway device with no internet access, the gateway waits until it indicates that there is no available Cloud connection. When running offline, data telemetry for the location is lost. Also, the mobile app cannot be used. Despite being offline, the ability to use LAN communication allows the users to troubleshoot the internet connection.

OVSDB Interfaces

Device freeze restricts client access to the internet or local network[4]. Users set the rules over mobile app/cloud or local GUI if present. The user can perform instant freeze or schedule freeze (Bedtime, School nights, custom).

[4] The Device Freeze functionality is similar to what is known as “parental control” elsewhere.

Managed OVSDB tables:

- Client_Nickname_Config (nickname synchronization)
- Client_Freeze_Config (device freeze)

Thermal management keeps the device temperature within operational limits to ensure uninterrupted performance. The device temperature drops by:

- increasing the airflow (increasing the fan revolutions)
- reducing the amount of generated heat (switching off the Wi-Fi antennas)

Thermal state is the state of the highest temperature of all radios. Thermal states table is compiled in the code. Each platform has its own thermal states table, but can be overridden by inserting values into the Node_Config OVSDB table.

Example thermal states table:

Thermal state	wifi0 temperature (℃)	wifi1 temperature (℃)	Desired fan speed (RPM)	wifi0 txchainmask	wifi1 txchainmask
0	0	0	0	3	15
1	65	73	3500	3	15
2	69	77	4500	3	15
3	71	80	5500	3	15
4	73	83	5500	3	7
5	76	86	5500	1	7
6	78	89	5500	1	7
7	81	92	5500	1	3
8	84	95	5500	1	3

If the temperatures of the Wi-Fi radios is 65, 85 (wifi0, wifi1 respectively), the calculated thermal state is 4, because the temperature for wifi1 (85) is higher than the temperature for state 4, but less than state 5. TM would in this state set the fan to 5500 RPM, and lower the *txchainmask* to 7 for wifi1.

TM applies the temperature hysteresis when calculating the correct thermal state. The node does not decrease the thermal state until the temperature drops for at least 3 degrees centigrade from a lower state.

Thermal Manager reboots the device if:

- Reducing Tx chainmasks does not reduce the device temperature despite running the fan at full speed
- The temperature remains above the highest thermal state temperature for 30 seconds

Device reboot causes the Wi-Fi chips and the main SoC to be reset, which lowers heat generation.

Managed OVSDB table for thermal management:

- Node_Config

Thermal states table (which is compiled into the Platform Manager code) can be overridden by writing the fields into the Node_Config OVSDB table. This is a key-value storage table.

Keys that are valid and checked by the TM are:

Key	Description
SPFAN_state[*state*]_wifi[r*adio_num*]_temp	Defines temperatures for each Wi-Fi radio and for each state.
SPFAN_state[*state*]_wifi[r*adio_num*]_txchainmask	Defines *txchainmask* for each Wi-Fi radio and for each state.
SPFAN_state[*state*]_fanrpm	Defines fan speed (in RPM) for each state.

Example: To override temperatures for thermal state 2 to values 100, 101 with fan RPM of 6000, the following values should be written to the Node_Config table:

```
ovsdb-client transact '["Open_vSwitch",{"op":"insert", "table":"Node_Config",
"row":{"key":"SPFAN_state2_wifi0_temp","value":"100", "module":"tm"} }]'

ovsdb-client transact '["Open_vSwitch",{"op":"insert", "table":"Node_Config",
"row":{"key":"SPFAN_state2_wifi1_temp","value":"101", "module":"tm"} }]'

ovsdb-client transact '["Open_vSwitch",{"op":"insert", "table":"Node_Config",
"row":{"key":"SPFAN_state2_fanrpm","value":"6000", "module":"tm"} }]'
```

LED management sets up and controls the device LED operation.

Managed OVSDB table for LED management:

- AWLAN_Node

Log management collects and sends the system log information upon Cloud request to a predefined location.

Managed OVSDB table for Log management:

- AW_LM_Config

> ***Note:*** *Having the log management functionality integrated within the PM optimizes device resource usage, especially RAM.*

Target Library Interface

The PM features use the below listed target functions.

Client freeze:

- target_client_freeze_register
- target_client_freeze_set
- target_client_nickname_register
- target_client_nickname_set

Thermal management:

- osp_tm_init
- osp_tm_deinit
- osp_tm_is_temp_src_enabled
- osp_tm_get_temp_src_name
- osp_tm_get_temperature
- osp_tm_get_fan_rpm

LED management:

- osp_led_init
- osp_led_set_state
- osp_led_clear_state
- osp_led_reset
- osp_led_get_state

Log management:

- target_log_pull

Exchange Manager - XM

Exchange manager facilitates data transfer between OVSDB and other management systems, such as TR-069/369[5]. It enables bi-directional synchronisation of selected attributes between two management system databases (e.g., OVSDB and TR-181).

Stub code is provided, where communication with an external database must be implemented to provide needed functionality.

It is recommended to use a single resource access path when both management systems are present. The two management systems should work in active/passive mode. When a management system is in active mode, it manages the resources (i.e. channel) and the other management system is in passive mode. This means that the "passive" management system only receives changes via XM into its database, but does not configure the resources.

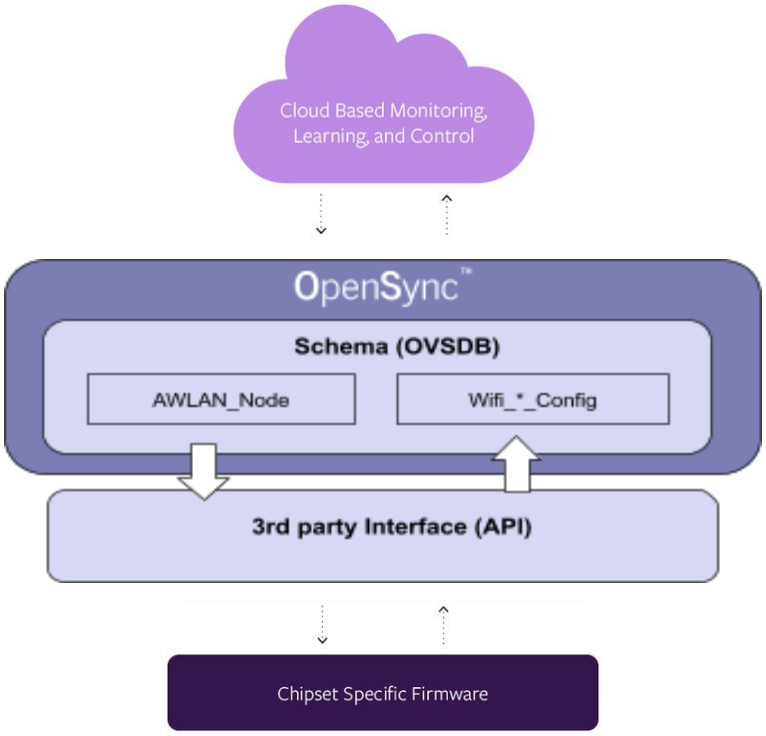


Figure 16: *The OpenSync Exchange Manager feature*

[5] Similar to RDK Mesh Agent (mesh-agent package), a native RDK component, the XM is responsible for synchronizing ACS managed parameters with *OpenSync* during operations.

OVSDB Interface

Vendor software has the ability to update basic device info such as model, serial number, etc., and also configure Wi-Fi or network settings.

Managed OVSDB tables

- AWLAN_Node
- Wifi_Radio_Config
- Wifi_VIF_Config
- Wifi_Inet_Config

Connector Interface

XM uses a target abstraction library called connector. These are the APIs:

- connector_init
- connector_close
- connector_sync_mode
- connector_sync_radio
- connector_sync_vif
- connector_sync_inet

In the initialization function, the XM passes a structure of callback functions, which are then used when handling changes from local GUI, TR-069, or other.

- connector_device_info_cb - Device info
- connector_device_mode_cb - Operation mode (monitor/cloud)
- connector_cloud_address_cb - Cloud address to connect to
- connector_radio_update_cb - Settings for each radio
- connector_vif_update_cb - Settings for each AP/STA interface
- connector_inet_update_cb - Settings for each eth/bridge with dhcp/ip

Upgrade Manager - UM

After receiving a trigger from the Cloud, the UM makes sure the device firmware upgrade process completes. Cloud orchestrates the firmware upgrade process for all nodes on the selected customer locations.

The Cloud initiates updates using:

- URL at which the firmware image is available
- Download time span during which the Cloud expects the image download
- Firmware password if the image is encrypted (optional)

After each successful image download, devices report success through the upgrade_status field of the OVSDB table (AWLAN_Node). At this point, the Cloud updates the upgrade timer. After the timer elapses, the devices start the upgrade procedure. If the upgrade is successful, the

devices again report success through the upgrade status field. The Cloud initiates device reboot right after.

Upgrade Process

AWLAN_Node	Direction	Comments
firmware_url	cloud→device	Download URL for firmware image and MD5 file.
firmware_pass	cloud→device	Password to decrypt image.
upgrade_timer	cloud→device	Configure firmware upgrade timer in seconds.
upgrade_dl_timeout	cloud→device	Defines download timeout in seconds.
upgrade_status	device→cloud	Reflects current upgrade status (*check values below*)

Before initiating the upgrade process, the following requirements must be met:

- The node (device) needs to be listed in the inventory database.
- The node must be online.
- The node must be onboarded and claimed to a location.
- Upgrade firmware version matrix must be provided with valid firmware details and the new image URL.
- The firmware image must be loaded to the AWS S3 data storage and accessible via URL indicated in the firmware version matrix.
- Current node firmware version must be different from the upgrade firmware version.

If the above conditions are met, and the upgrade was initialized, the Cloud starts instructing the UM to download the image while checking the download status in the *upgrade_status* field of the AWLAN_Node table. Possible response statuses are:

Value	Status	Description
10	UPG_STS_FW_DL_START	FW download started
11	UPG_STS_FW_DL_END	FW download successfully completed
20	UPG_STS_FW_WR_START	FW write on alt partition started
21	UPG_STS_FW_WR_END	FW image write successfully completed
30	UPG_STS_FW_BC_START	Bootconfig partition update started
31	UPG_STS_FW_BC_END	Bootconfig partition update completed
-1	UPG_ERR_ARGS	Wrong arguments (app error)
-3	UPG_ERR_URL	Incorrect URL

-4	UPG_ERR_DL_FW	Failed firmware image download
-5	UPG_ERR_DL_MD5	Error while downloading firmware md5 sum file
-6	UPG_ERR_MD5_FAIL	md5 checksum file error
-7	UPG_ERR_IMG_FAIL	Firmware image error
-8	UPG_ERR_FL_ERASE	Flash erase failed
-9	UPG_ERR_FL_WRITE	Flash write failed
-10	UPG_ERR_FL_CHECK	Flash verification failed
-11	UPG_ERR_BC_SET	Set new bootconfig failed
-12	UPG_ERR_APPLY	Device restart failed
-14	UPG_ERR_BC_ERASE	Flash BootConfig erase failed
-15	UPG_ERR_SU_RUN	Safe update is running
-16	UPG_ERR_DL_NOFREE	Not enough free space on device

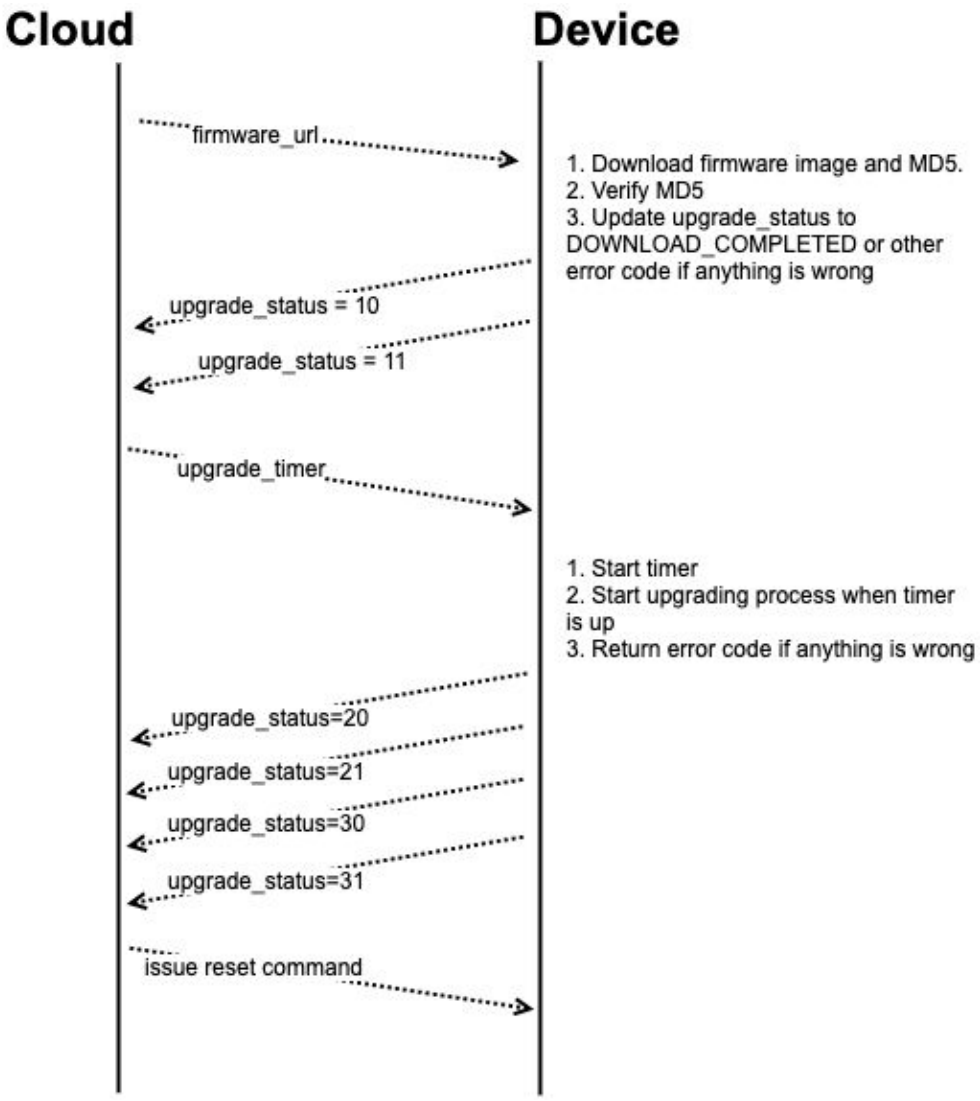


Figure 16: *The upgrade process flow*

Web-based Firmware Upgrades

UM can be used to upgrade the device firmware using the OSP API. This means that the firmware can be remotely upgraded using a web-based network management portal, such as Plume NOC.

OVSDB Interface

The OVSDB table used by the upgrade procedure is AWLAN_Node.

Field for communication from the Cloud to devices:

- firmware_pass
- firmware_url

- firmware_version
- upgrade_dl_timer
- upgrade_timer

Fields for communication from device to the Cloud:

- upgrade_status

Target Library Interface

APIs Used:

- osp_upg_check_system
- osp_upg_dl
- osp_upg_upgrade
- osp_upg_commit
- osp_upg_errno

List of libc API Calls

OpenSync runs a stack of API calls. The majority of these calls are standard POSIX and C functions. The standards are defined next to the functions. However, a portion of the API calls are non-standard functions – these are marked using the *is present in/not in standard* notation.

> ***Note:*** *The list of API calls depends on the current release and specific deployment, and may not include all available functions.*

- abort: SVr4, POSIX.1-2001, POSIX.1-2008, 4.3BSD, C89, C99
- abs: POSIX.1-2001, POSIX.1-2008, C99, SVr4, 4.3BSD. C89
- accept: POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD
- access: SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008
- alloca: *not in POSIX.1*, appeared in 32V, PWB, PWB.2, 3BSD, and 4BSD, GNU
- assert: POSIX.1-2001, POSIX.1-2008, C89, C99
- atexit: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- atof: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- atoi: POSIX.1-2001, POSIX.1-2008, C99, SVr4, 4.3BSD, C89 and POSIX.1-1996
- atol: POSIX.1-2001, POSIX.1-2008, C99, SVr4, 4.3BSD. C89 and POSIX.1-1996
- basename: POSIX.1-2001, POSIX.1-2008
- bind: POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD
- bsearch: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- bzero: LEGACY in POSIX.1-2001, POSIX.1-2008 removes the specification, memset() should be used in new programs
- calloc: POSIX.1-2001, POSIX.1-2008, C89, C99
- cfsetispeed: POSIX.1-2001
- cfsetospeed: POSIX.1-2001
- chdir: POSIX.1-2001
- clock_gettime: POSIX.1-2001, POSIX.1-2008, SUSv2
- clock_nanosleep: POSIX.1-2001, POSIX.1-2008
- close: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- closedir: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- connect: POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD
- dirname: POSIX.1-2001, POSIX.1-2008
- difftime: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- dladdr: *is present in glibc 2.0 and later*
- dlclose: POSIX.1-2001
- dlopen: POSIX.1-2001
- dlerror: POSIX.1-2001
- dlsym: POSIX.1-2001
- dup2: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- encrypt: POSIX.1-2001, POSIX.1-2008, SUS, SVr4
- ether_aton: *is present in 4.3BSD, SunOS*
- execl: POSIX.1-2001, POSIX.1-2008
- execlp: POSIX.1-2001, POSIX.1-2008
- execv: POSIX.1-2001, POSIX.1-2008

- execve: POSIX.1-2001, POSIX.1-2008
- exit: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- execvp: POSIX.1-2001, POSIX.1-2008
- execvpe: *GNU extension*
- fcntl: SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008
- fdopen: POSIX.1-2001, POSIX.1-2008
- feof: C89, C99, POSIX.1-2001, and POSIX.1-2008
- fflush: C89, C99, POSIX.1-2001, POSIX.1-2008
- fgets: POSIX.1-2001, POSIX.1-2008, C89, C99
- ferror: C89, C99, POSIX.1-2001, and POSIX.1-2008
- fgets: POSIX.1-2001, POSIX.1-2008, C89, C99
- fileno: POSIX.1-2001 and POSIX.1-2008
- flock: *4.4BSD, appears on most UNIX systems*
- fnmatch: POSIX.1-2001, POSIX.1-2008, POSIX.2
- fopen: POSIX.1-2001, POSIX.1-2008, C89, C99
- fork: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- fputc: POSIX.1-2001, POSIX.1-2008, C89, C99
- fprintf: POSIX.1-2001, POSIX.1-2008, C89, C99
- fputs: POSIX.1-2001, POSIX.1-2008, C89, C99
- fread: POSIX.1-2001, POSIX.1-2008, C89
- free: POSIX.1-2001, POSIX.1-2008, C89, C99
- freeaddrinfo: POSIX.1-2001, POSIX.1-2008
- fscanf: C89 and C99 and POSIX.1-2001 (some modifiers may behave differently)
- fseek: POSIX.1-2001, POSIX.1-2008, C89, C99
- fstat: SVr4, 4.3BSD, POSIX.1-2001, POSIX.1.2008
- fsync: POSIX.1-2001, POSIX.1-2008, 4.3BSD
- ftell: POSIX.1-2001, POSIX.1-2008, C89, C99
- ftruncate: POSIX.1-2001, POSIX.1-2008, 4.4BSD, SVr4
- fwrite: POSIX.1-2001, POSIX.1-2008, C89
- gai_strerror: POSIX.1-2001, POSIX.1-2008
- getaddrinfo: POSIX.1-2001, POSIX.1-2008
- getenv: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- gethostname: SVr4, 4.4BSD, POSIX.1-2001 and POSIX.1-2008
- getnameinfo: POSIX.1-2001, POSIX.1-2008, RFC 2553
- getopt: POSIX.1-2001, POSIX.1-2008, and POSIX.2
- getopt_long_only: *GNU extensions*
- getopt_long: *GNU extensions*
- getpeername: POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD
- getpagesize: SVr4, 4.4BSD, SUSv2 (marked as legacy), POSIX.1-2001 *dropped* it
- getsockopt: POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD
- getpid: POSIX.1-2001, POSIX.1-2008, 4.3BSD, SVr4
- gettimeofday: SVr4, 4.3BSD. POSIX.1-2001, POSIX.1-2008 *marks as obsolete, recommends clock_gettime instead*
- glob: POSIX.1-2001, POSIX.1-2008, POSIX.2
- globfree: POSIX.1-2001, POSIX.1-2008, POSIX.2.
- gmtime: POSIX.1-2001. C89 and C99
- gmtime_r: POSIX.1-2001. C89 and C99
- htonl: POSIX.1-2001, POSIX.1-2008

- htons: POSIX.1-2001, POSIX.1-2008
- if_indextoname: POSIX.1-2001, POSIX.1-2008, RFC 3493
- if_nametoindex: POSIX.1-2001, POSIX.1-2008, RFC 3493
- index: 4.3BSD; marked as LEGACY in POSIX.1-2001. *POSIX.1-2008 removes it, advises strchr() and strrchr()*
- inet_addr: POSIX.1-2001, POSIX.1-2008, 4.3BSD
- inet_aton: *not specified in POSIX.1, but available on most systems*
- inet_ntoa: POSIX.1-2001
- inet_ntop: POSIX.1-2001, POSIX.1-2008
- inet_pton: POSIX.1-2001, POSIX.1-2008
- ioctl: *No single standard. Arguments, returns, and semantics of ioctl() vary*
- isalnum: C89
- isalpha: C89
- isascii: POSIX.1-2001, *POSIX.1-2008 marks as obsolete*
- isatty: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- isdigit: C89
- islower: C89
- isgraph: C89
- isprint: C89
- isupper: C89
- isspace: C89
- iswdigit: POSIX.1-2001, POSIX.1-2008, C99
- isxdigit: C89
- kill: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- listen: POSIX.1-2001, POSIX.1-2008, 4.4BSD
- localtime: POSIX.1-2001. C89 and C99
- longjmp: POSIX.1-2001, POSIX.1-2008, C89, C99
- lstat: SVr4, 4.3BSD, POSIX.1-2001, POSIX.1.2008
- lseek: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- malloc: POSIX.1-2001, POSIX.1-2008, C89, C99
- memchr: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- memcmp: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- memcpy: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- memmove: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- memmem: *not specified in POSIX.1* (but available on many systems)
- memset: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- memrchr: *GNU extension*
- mkstemp: 4.3BSD, POSIX.1-2001
- mktime: POSIX.1-2001. C89 and C99
- mkdir: POSIX.1-2008
- mmap: POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD
- modf: C99, POSIX.1-2001, POSIX.1-2008
- munmap: POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD
- nanosleep: POSIX.1-2001, POSIX.1-2008
- ntohl: POSIX.1-2001, POSIX.1-2008
- ntohs: POSIX.1-2001, POSIX.1-2008
- open: SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008
- opendir: SVr4, 4.3BSD, POSIX.1-2001

- openlog: SUSv2, POSIX.1-2001, and POSIX.1-2008
- pclose: POSIX.1-2001, POSIX.1-2008
- perror: POSIX.1-2001, POSIX.1-2008, C89, C99, 4.3BSD
- pipe: POSIX.1-2001, POSIX.1-2008
- poll: POSIX.1-2001 and POSIX.1-2008
- popen: POSIX.1-2001, POSIX.1-2008
- pow: C99, POSIX.1-2001, POSIX.1-2008
- prctl: *Linux-specific*
- printf: POSIX.1-2001, POSIX.1-2008, C89, C99
- pthread_create: POSIX.1-2001, POSIX.1-2008
- pthread_detach: POSIX.1-2001, POSIX.1-2008
- pthread_mutex_destroy: POSIX.1-2008
- pthread_mutex_init: POSIX.1-2008
- pthread_mutex_lock: POSIX.1-2008
- pthread_mutex_unlock: POSIX.1-2008
- putc: POSIX.1-2001, POSIX.1-2008, C89, C99
- putchar: POSIX.1-2001, POSIX.1-2008, C89, C99
- qsort: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- rand: SVr4, 4.3BSD, C89, C99, POSIX.1-2001
- raise: POSIX.1-2001, POSIX.1-2008, C89, C99
- read: SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008
- random: SVr4, 4.3BSD, POSIX.1-2001
- readdir: *Linux-specific*
- readlink: 4.4BSD, POSIX.1-2001, POSIX.1-2008
- realloc: POSIX.1-2001, POSIX.1-2008, C89, C99
- realpath: 4.4BSD, POSIX.1-2001
- recv: 4.4BSD, POSIX.1-2001
- recvfrom: POSIX.1-2001, POSIX.1-2008, 4.4BSD
- regexec: POSIX.1-2001, POSIX.1-2008
- regcomp: POSIX.1-2001, POSIX.1-2008
- regex: POSIX.1-2001, POSIX.1-2008
- remove: POSIX.1-2001, POSIX.1-2008, C89, C99, 4.3BSD
- rename: 4.3BSD, C89, C99, POSIX.1-2001, POSIX.1-2008
- res_init: *4.3BSD*
- rewind: POSIX.1-2001, POSIX.1-2008, C89, C99
- sem_init: POSIX.1-2001
- select: POSIX.1-2001, POSIX.1-2008, and 4.4BSD
- send: 4.4BSD, SVr4, POSIX.1-2001
- sem_post: POSIX.1-2001
- sem_wait: POSIX.1-2001
- sendto: 4.4BSD, SVr4, POSIX.1-2001
- setenv: POSIX.1-2001, POSIX.1-2008, 4.3BSD
- setpgid: POSIX.1-2001
- setjmp: POSIX.1-2001, POSIX.1-2008, C89, C99
- setsid: POSIX.1-2001, POSIX.1-2008, SVr4
- getsockopt: POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD
- sigaction: POSIX.1-2001, POSIX.1-2008, SVr4
- sigemptyset: POSIX.1-2001, POSIX.1-2008

- signal: POSIX.1-2001, POSIX.1-2008, C89, C99
- sleep: POSIX.1-2001, POSIX.1-2008
- snprintf: POSIX.1-2001, POSIX.1-2008, C99
- socket: POSIX.1-2001, POSIX.1-2008, 4.4BSD
- sprintf: POSIX.1-2001, POSIX.1-2008, C89, C99
- srand: SVr4, 4.3BSD, C89, C99, POSIX.1-2001
- srandom: POSIX.1-2001, POSIX.1-2008, 4.3BSD
- sscanf: C89 and C99 and POSIX.1-2001
- stat: SVr4, 4.3BSD, POSIX.1-2001, POSIX.1.2008
- statvfs: POSIX.1-2001, POSIX.1-2008
- strcasestr: *non-standard extension*
- strcasecmp: 4.4BSD, POSIX.1-2001, POSIX.1-2008
- strcat: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strchr: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strcpy: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strcspn: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strcmp: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strdup: SVr4, 4.3BSD, POSIX.1-2001
- strdupa: *non-standard extension* (a macro)
- strerror: POSIX.1-2001, POSIX.1-2008, C89, and C99
- strftime: SVr4, C89, C99
- strlen: POSIX.1-2001, POSIX.1-2008, C89, C99, C11, SVr4, 4.3BSD
- strncasecmp: 4.4BSD, POSIX.1-2001, POSIX.1-2008
- strncat: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strncmp: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strncpy: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strndup: POSIX.1-2008
- strnlen: POSIX.1-2008
- strpbrk: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strptime: POSIX.1-2001, POSIX.1-2008, SUSv2
- strrchr: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strsep: *4.4BSD*
- strsignal: POSIX.1-2008
- strspn: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strtod: POSIX.1-2001, POSIX.1-2008, C99
- strstr: POSIX.1-2001, POSIX.1-2008, C89, C99
- strtok_r: POSIX.1-2001, POSIX.1-2008
- strtok: POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD
- strtoll: POSIX.1-2001, POSIX.1-2008, C99
- strtoul: POSIX.1-2001, POSIX.1-2008, C89, C99 SVr4
- strtoull: POSIX.1-2001, POSIX.1-2008, C99
- strtol: POSIX.1-2001, POSIX.1-2008, C89, C99 SVr4, 4.3BSD
- sync: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- syscall: *GNU extension*
- sysconf: POSIX.1-2001, POSIX.1-2008
- sysinfo: *Linux-specific*
- syslog: *Linux-specific*
- system: POSIX.1-2001, POSIX.1-2008, C89, C99

- tcdrain: POSIX.1-2001
- tcflush: POSIX.1-2001
- tcgetattr: POSIX.1-2001
- tcsetattr: POSIX.1-2001
- time: SVr4, 4.3BSD, C89, C99, POSIX.1-2001
- timegm: *GNU extensions* (should be avoided)
- tolower: C89, C99, 4.3BSD, POSIX.1-2001, POSIX.1-2008
- toupper: C89, C99, 4.3BSD, POSIX.1-2001, POSIX.1-2008
- umask: POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD
- unlink: SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008
- usleep: 4.3BSD, POSIX.1-2001 (*obsolete should use nanosleep()*)
- vfprintf: POSIX.1-2001, POSIX.1-2008, C89, C99
- vprintf: POSIX.1-2001, POSIX.1-2008, C89, C99
- vsnprintf: POSIX.1-2001, POSIX.1-2008, C99
- vsprintf: POSIX.1-2001, POSIX.1-2008, C89, C99
- wait: SVr4, 4.3BSD, POSIX.1-2001
- waitpid: SVr4, 4.3BSD, POSIX.1-2001
- write: SVr4, 4.3BSD, POSIX.1-2001

Compiler built-ins:

- __builtin_popcount *GCC specific*
- __builtin_expect *GCC specific*

Cloud Requirements

When connecting your gateway to the cloud using *OpenSync*, you will want to work with Plume on several topics, such as device certificates for server-side client authentication, the Cloud deployment account, user management options, etc. The Cloud also expects some device parameters to be known in advance, prior to first connection: device ID (serial number), LAN and WAN interface MAC address, Wi-Fi radio MAC addresses.

Device Certificates

Connecting devices with *OpenSync* to the Cloud requires client certificates in order to guarantee client authenticity. the Cloud strictly enforces client authentication during all TLS connections to clients. Establishing the connection between the cloud and the device will perform mutual authentication, therefore on the device side *OpenSync* needs access to the device certificate and the cloud certificate of authority. The encryption and storage of the certificates is tailored per device depending on capabilities (dedicated crypto hardware, etc.).

Device Capabilities

The Cloud makes certain decisions based on the *OpenSync* model and firmware version reported by the device. These decisions include supported features, wifi capabilities such as the number of radios, number of spatial streams, supported channels, transmit power, network interface names, etc.

Troubleshooting

To help with the integration process, *OpenSync* provides functionality to manually interact with OVSDB tables to either monitor or to alter content. This tool is similar to generic ovsdb tools like *ovsdb-client*. This tool is also used to dynamically change the logging verbosity level of specified managers in order to observe the flow of events in more detail.

ovsh tool

OpenSync relies heavily on ovsdb functionality and having a tool that makes it easier to manipulate table contents often comes in handy during the integration phase or for debugging during device operation.

The *ovsh* tool uses the jansson library as a JSON parsing and manipulating library and itself provides a user-friendly command line interface. Here is the usage:

```
ovsh    COMMAND    TABLE    [-w|--where    [COLUMN==VAL|COLUMN!=VAL]]    [COLUMN[:=VAL]]    [COLUMN[::OBJ]]
[COLUMN[~=STRING]]...

COMMAND can be one of:
   - s/select: Read one or multiple rows
   - i/insert: Insert a new row
   - u/update: Update one or multiple rows
   - d/delete: Delete one or multiple rows
   - w/wait:   Perform a WAIT operation on one or multiple rows

TABLE specifies an OVS table; this is a required option.

Additional options are:
   -w EXPR | --where=EXPR
                - Use a WHERE statement to filter rows (select, update, wait)
   -j|--json    - print the result in JSON format (select, insert, update)
   -T|--table   - print the result in table format (select)
   -r|--raw     - print the result in RAW format (select)
   -c|--column  - print the result in single COLUMN (select)
   -m|--multi-line - print the result in multi COLUMNS, split long lines [default] (select)
   -M|--multi   - print the result in multi COLUMNS, do not split long lines (select)
   -u|--uuid-compact     - print compact uuid [default] (select)
   -U|--no-uuid-compact - do not print compact uuid (select)
   -a|--abbrev  - abbreviate uuid to 1234~6789 [default] (select)
   -A|--no-abbrev  - do not abbreviate uuid (select)
   -o fmt | -output=fmt
                - print the result in a custom printf-like format (select)
   -t MSEC | --timeout=MSEC
                - The timeout in milliseconds for a wait command
   -n|--notequal   - Use the "!=" (not-equal) operator instead of "==" (wait)
   -v|--verbose  - Increase debugging level
   -q|--quiet    - Report only errors
```

Note: *print modes: json, raw and custom do not compact and abbreviate uuid*

A common example of use is an *update* command, followed by a *select* command to verify the transaction:

```
# /usr/plume/tools/ovsh s SSL
-----------------------------------------------
_uuid             | 1234~abcd                 |
_version          | efgh~5678                 |
bootstrap_ca_cert | false                     |
ca_cert           | none                      |
certificate       | /var/certs/client.pem     |
external_ids      | ["map",[]]                |
private_key       | /var/certs/client_dec.key |
-----------------------------------------------
# /usr/plume/tools/ovsh u SSL ca_cert:=/var/certs/ca.pem
1

# /usr/plume/tools/ovsh s SSL
-----------------------------------------------
_uuid             | 1234~abcd                 |
_version          | efgh~5678                 |
bootstrap_ca_cert | false                     |
ca_cert           | /var/certs/ca.pem         |
certificate       | /var/certs/client.pem     |
external_ids      | ["map",[]]                |
private_key       | /var/certs/client_dec.key |
-----------------------------------------------
```

Debug Tracing Level

To change log severity for a specific module in runtime one must use the OVSDB interface. To do that you need to insert/update a row in the *AW_Debug* table with appropriate **name** and **log_severity** (see AW_Debug table for exact description). Normally this can be done using the “ovsh” tool.

Requirements:

- OVSDB server and Log Manager must be operational
- Module needs to be registered with *OpenSync* logging library to receive updates (see target_log_open() and log_register_dynamic_severity())

Example where we set log severity to TRACE level for Statistics Manager, identified with SM module ID:

```
root@OpenWrt:~# /usr/plume/tools/ovsh insert AW_Debug name:=SM log_severity:=TRACE      # Set
5a50400a-885a-445c-91d3-2f6a37ebdf9b
root@OpenWrt:~# /usr/plume/tools/ovsh select AW_Debug                                  # Verify
-----------------------------
_uuid           | 5a50~df9b |
_version        | e1cb~01bb |
log_severity    | TRACE     |
name            | SM        |
-----------------------------
```

OpenSync modules that support dynamic log severity changes:

Module Name	Module ID
Diagnostics Manager	DM
Connection Manager	CM
Wireless Manager	WM
Network Manager	NM
Statistics Manager	SM
OpenFlow Manager	OM
Log Manager	LM
Band Steering Manager	BM
Queue Manager	QM
Platform Manager	PM
Upgrade Manager	UM
Exchange Manager	XM
Flow Collection Manager	FCM
Flow Service Manager	FSM