OpenSync™

RDK-B OpenSync Integration

DATE: April 08, 2020

Document number: 019-1129-20

History of Changes

Version	Change
November 29, 2019	Release 1.4.0
March 9, 2020	Release 1.4.0.2
April 8, 2020	Update wifihal version to 2.16

Table of Contents

References

[1] *https://www.opensync.io/documentation/*

Introduction

OpenSync™ is designed to provide a software defined network - SDN platform, through which it virtualizes the networking and wireless management for easy service roll-out. It acts as a silicon, CPE, and cloud-agnostic connection between in-home hardware devices and the cloud. It provides a modern set of utilities for collecting measurement and other telemetry data from devices. It also enables remote control and management of the devices, and advanced capabilities for specific services, including Wi-Fi meshing, access control, cybersecurity, parental controls, and IoT onboarding and telemetry. Services/Features can be added/removed and controlled directly from the cloud - instead of adding them via a lengthy FW upgrade process.

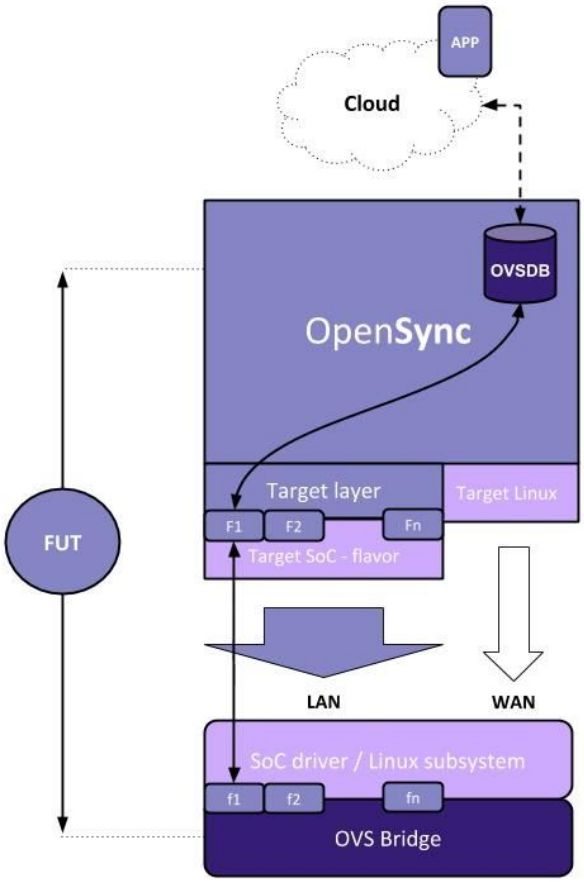

***Figure 1**: OpenSync block diagram*

OpenSync provides a range of benefits and features to chipset suppliers, system integrators, and operators:

- Enables rapid development and deployment of new services and products
- Defines open, interoperable, multi-vendor interfaces, at multiple levels, that have been adopted broadly in the industry
- Provides efficient methods for telemetry and control, based on modern open industry solutions

- Supports and provides utilities for a broad range of existing services
- Is easily extensible, allowing the addition of new services, typically with only cloud software changes
- Is proven, robust, and already deployed widely.

Devices that can support *OpenSync* are referred to as **targets,** where the **target layer** is an adaptation layer between the *OpenSync* managers and the low-level SoC/Linux drivers.

There are many different **target layer flavors,** which can be specific to a particular chipset (e.g. Broadcom, Qualcomm, Quantenna, Celeno, etc.), or a platform such as RDK, OpenWrt, PRPL, etc.

This document describes the integration of *OpenSync* with RDK. Details regarding *OpenSync* can be found at *https://www.opensync.io/documentation/ [1]*.

For more information on RDK, visit RDKCentral.com.

OpenSync Target Layer

OpenSync RDK target layer flavor adapts *OpenSync* to the RDK software stack.

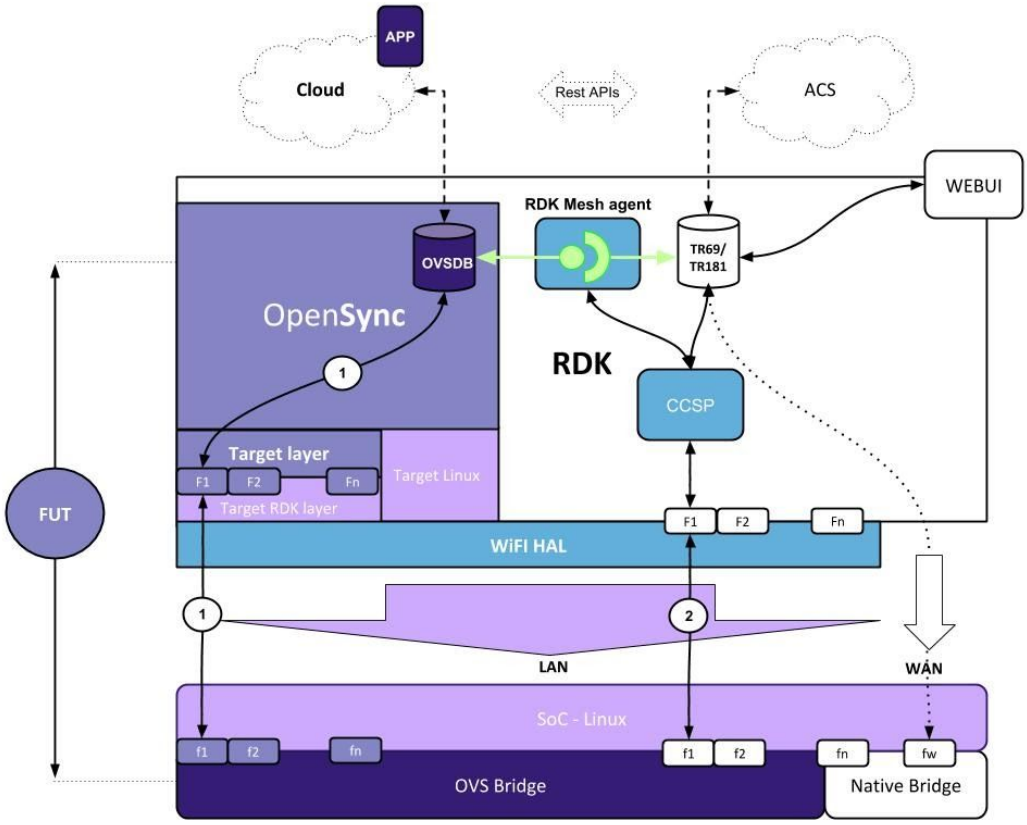

***Figure 2**: OpenSync RDK layer*

The following connections into the RDK platform are used:

- Logger for linking *OpenSync* logging system to the RDK Logger
- Linux networking utilities for managing VLANs, GRETAPs, and Bridges
- Device Info for entity information (*deviceinfo.sh* in RDK-B)
- RDK MeshAgent for synchronizing configuration changes
- Wi-Fi HAL for SoC interaction such as steering, statistics, Wi-Fi management

OpenSync code can be downloaded from OpenSync Source Code repository.

RDK-B Requirements

This section provides a list of RDK-B requirements to support the *OpenSync* RDK-B component.

Internet Access

The processor which runs the *OpenSync* will require Internet access for connectivity to the Plume cloud. This includes outbound TCP connections on port 443, and DNS resolution.

Device Information and Configuration

The *OpenSync* component expects the RDK-B installation to include a script called ***deviceinfo.sh***, which is used to query entity information and *OpenSync* configuration controlled by RDK-B.

```
Usage:deviceinfo.sh
[-mo|-sn|-fw|-cmac|-cip|-cipv6|-emac|-eip|-eipv6|-lmac|-lip|-lipv6|-ms|-mu]
```

Option	Description
-mo	Model number of device, e.g. TG1682G
-sn	Serial number of device, e.g. 12345678
-fw	Firmware version of device, e.g. TG1682_2.8p15s1_PROD_sey
-cmac	CM MAC address, e.g. 01:02:AA:BB:CC:DD
-cip	CM IPv4 assigned address
-cipv6	CM IPv6 assigned address
-emac	Erouter MAC address
-eip	Erouter IPv4 assigned address
-eipv6	Erouter IPv6 assigned address
-lmac	LAN MAC address
-lip	LAN IPv4 assigned address
-lipv6	LAN IPv6 assigned address
-ms	Mesh state: Active or Passive
-mu	Mesh URL for Plume cloud, e.g. wildfire.plume.tech

The CM MAC provided using ***-cmac*** is used as the unique identifier for every gateway when connecting to the Plume cloud. When referencing the gateway in the Plume cloud, such as to claim it to a given account, the CM MAC -- all capitals with no colons -- is what is used.

The Mesh State defines what state the *OpenSync* will operate in. Currently two states are supported:

- **Passive:** In this state, the *OpenSync* will only collect Wi-Fi statistics and report them to the Plume cloud. There are no write operations such as changing the channel or channel width. The *OpenSync* backhaul is also not enabled, so extenders are not able to connect.
- **Active:** In this state, the *OpenSync* will operate in full control mode. In this mode it will gather statistics, control the channel and channel width, and will also bring up the *OpenSync* backhaul, allowing extenders claimed to the same Plume cloud account to connect and extend the gateway's SSIDs.

The Mesh URL defines which cloud environment the gateway should connect to. For initial proof-of-concept work and general development, the Plume 'dev' redirector address should be used: ***wildfire.plume.tech***

OpenSync Backhaul Support

The *OpenSync* backhaul that is used for *OpenSync* enabled extender connectivity requires two wireless interfaces, as well as a bridge and a subnet that can be used by the extenders to gain internet access for cloud connectivity.

Wireless Requirements

RDK-B must provide SSID index 12 (vif12), and SSID index 13 (vif13) for use by the *OpenSync*. These SSIDs should be disabled by default at boot time, and are enabled upon request by the *OpenSync* using wifi_hal APIs described later in this document. For example:

- **vif12** (*ssid index 12*): Used for 2.4G OpenSync enabled extender connections
    - IP address of 169.254.0.1/24 must be assigned
    - DHCP must assign IP's within the 169.254.0.2 - 169.254.0.254 range
    - MTU must be 1600

- **vif13** (*ssid index 13*): Used for 5G OpenSync enabled extender connections
    - IP address of 169.254.1.1/24 should be assigned
    - DHCP must assign IP's within the 169.254.1.2 - 169.254.1.254 range
    - MTU must be 1600

The link-local networks assigned (169.254.0.0/24 and 169.254.1.0/24) are only used for GRE tunneling, and do not require any external internet access.

Backhaul Bridge and Subnet

The processor running the *OpenSync* must have an existing bridge, which is to be used for Internet connectivity from the *OpenSync* enabled extenders. This bridge should have an IP address assigned to it, and DHCP should be running to hand out DHCP leases within the same subnet, providing default gateway and DNS options.

When extenders connect to the gateway, the *OpenSync* will automatically create a GRE tunnel, which will then be added into this bridge. The extenders will use the assigned IP address, default gateway, and DNS server to reach the Plume cloud. This outbound traffic will be limited to port 443 and DNS queries only.

Linux Networking

The *OpenSync* managers using the RDK-B adaption layer included in the RDK-B *OpenSync* component will use existing RDK-B tools to maintain the required GRE tunnels, 802.1q VLAN interfaces, and to manage their membership in pre-defined bridges.

Starting with *OpenSync* version 1.4, such native utilities are no longer required. A new HAL layer has been introduced, called *osync_hal*, whose role is to abstract the use of file system utilities etc. For function prototypes and details check the *osync_hal.h* header file.

Hardware Acceleration

Depending on where and how any hardware acceleration is tied into the system, it may require changes from the vendor. GRE is a well defined protocol, and is often used in the Enterprise sector. Most vendors that have some form of hardware acceleration already have support for GRE.

DHCP Lease Information

The *OpenSync* needs to monitor DHCP lease information, so that it can get information such as client assigned IP addresses, hostnames, and fingerprint data.

Currently this is designed to monitor the ***dnsmasq.leases*** file on the processor that is running the *OpenSync*. If this file is not available on the same processor, then we suggest running a synchronization program that will detect changes and copy them over to the processor running the *OpenSync*.

Starting with OpenSync version 1.4, implementation can be abstracted via *osync_hal* functions.

Networking Configuration

Before compiling the *OpenSync*, you must provide a networking map that includes information on which SSIDs should be extended through the *OpenSync* enabled extenders, along with their corresponding VLAN IDs and bridges. Here is an example:

```
static ifmap_t  ifmap[] = {
// idx   plume-ifname     dev-ifname   bridge   gre-br  vlan      description
//-------------------------------------------------------------------------------
 { 1,    "bhaul-ap-24",   "ath12",    "br201",  NULL,   0    },// 2G Backhaul
 { 1,    "bhaul-ap-50",   "ath13",    "br201",  NULL,   0    },// 5G Backhaul
 { 2,    "home-ap-24",    "ath0",     "br0",    NULL,   100  },// 2G User SSID
 { 2,    "home-ap-50",    "ath1",     "br0",    NULL,   100  },// 5G User SSID
 { 4,    "svc-d-ap-24",   "ath2",     "br1",    NULL,   101  },// 2G Video SSID
 { 4,    "svc-d-ap-50",   "ath3",     "br1",    NULL,   101  },// 5G Video SSID
 // Bridge mappings
 { 0,    "br-home",       "br0",      "br0",    NULL,   0    },// User Bridge
 { 0,    NULL,            NULL,        NULL,    NULL,   0    }
};
```

This table helps map interfaces, bridges, and VLANs on the gateway, and is used when extending multiple SSIDs through the *OpenSync* enabled extenders. Traffic stays separated using VLANs between the gateway and extenders, ensuring the same level of security.

NOTE: *Although the "target_map_ifname" API is still present in OpenSync release 1.4, it is now deprecated. Vendors are encouraged to provide interface information along with other device information (capabilities, optimizer attributes), which is then used to create a device profile on the Cloud (see also OpenSync Cloud Requirements).*

RDK-B Mesh Agent

The RDK-B Mesh Agent is responsible for providing the mesh TR-181 data model, starting and stopping the *OpenSync*, and using a sync protocol to exchange messages with the *OpenSync*. This helps keep data models in sync when lower-layer Wi-Fi parameters are changed by the Plume cloud, and provides connection events for things like Ethernet or MoCA clients.

An example of the TR-181 mesh data model:

```
Parameter    1 name:
    Device.DeviceInfo.X_RDKCENTRAL-COM_xOpsDeviceMgmt.Mesh.Enable
                type:       bool,    value: true
Parameter    2 name:
    Device.DeviceInfo.X_RDKCENTRAL-COM_xOpsDeviceMgmt.Mesh.URL
                type:     string,    value: ssl:wildfire.plume.tech:443
Parameter    3 name:
    Device.DeviceInfo.X_RDKCENTRAL-COM_xOpsDeviceMgmt.Mesh.State
                type:     string,    value: Full
Parameter    4 name:
    Device.DeviceInfo.X_RDKCENTRAL-COM_xOpsDeviceMgmt.Mesh.Status
                type:     string,    value: Full
```

*NOTE: The Mesh.URL and Mesh.State data model values configured are what is provided to the OpenSync through the "**deviceinfo.sh -mu**" and "**deviceinfo.sh -ms**" commands described above.*

RDK-B Mesh Agent is open sourced and is available under:
https://code.rdkcentral.com/r/plugins/gitiles/rdkb/components/opensource/ccsp/MeshAgent/

RDK Logger

The *OpenSync* will log all of its messages through RDK Logger, using the module "***LOG.RDK.MeshService***".

Current RDK-B configuration logs this to */rdklogs/logs/MeshServiceLog.txt.0*

RDK-B wifi_hal Requirements

The current minimum version of wifi_hal supporting the *OpenSync* is **v2.16**. These APIs are used for five primary functions: reading and setting Wi-Fi information, Wi-Fi statistics, steering, and DFS. Note that this document only provides a list of the APIs themselves. Their documentation and actual definition are outside the scope of this document, and can be found in places such as the *wifi_hal.h* definition.

Reading Wi-Fi Information

The following information is read through wifi_hal using the APIs listed below:

- HAL API Version
- Number of Radio entries
- Radio Information:
    - Ifname
    - Operating frequency band
    - Country Code
    - Support channels
    - Enable status
    - Auto channel enable status
    - Current channel
    - Current channel width
    - Current Transmit Power
    - Current minimum standard/mode (11b, 11g, 11n, 11a, 11ac, etc)
- SSID Information:
    - Ifname
    - Radio Index
    - Enabled status
    - Configured ESSID
    - Current/active ESSID
    - SSID broadcast status
    - Encryption settings (type, passphrase)
    - ACL configuration (mode, mac list)
    - Bridge info
    - AP Isolation
    - Associated client information and events

The following wifi_hal APIs are used to query various Wi-Fi information and settings:

- `INT wifi_getHalVersion(CHAR *output_string);`
- `INT wifi_getRadioNumberOfEntries(ULONG *output);`
- `INT wifi_getRadioIfName(INT radioIndex, CHAR *output_string);`
- `INT wifi_getRadioOperatingFrequencyBand(INT radioIndex, CHAR *output_string);`
- `INT wifi_getSSIDNumberOfEntries(ULONG *output);`

- `INT wifi_getApName(INT apIndex, CHAR *output_string);`
- `INT wifi_getSSIDRadioIndex(INT ssidIndex, INT *radioIndex);`
- `INT wifi_getApNumDevicesAssociated(INT apIndex, ULONG *output_ulong);`
- `INT wifi_getAssociatedDeviceDetail(INT apIndex, INT devIndex, wifi_device_t *output_struct);`
- `INT wifi_getRadioEnable(INT radioIndex, BOOL *output_bool);`
- `INT wifi_getRadioChannel(INT radioIndex,ULONG *output_ulong);`
- `INT wifi_getRadioAutoChannelEnable(INT radioIndex, BOOL *output_bool);`
- `INT wifi_getRadioTransmitPower(INT radioIndex, ULONG *output_ulong);`
- `INT wifi_getRadioCountryCode(INT radioIndex, CHAR *output_string);`
- `INT wifi_getRadioStandard(INT radioIndex, CHAR *output_string, BOOL *gOnly, BOOL *nOnly, BOOL *acOnly);`
- `INT wifi_getRadioPossibleChannels(INT radioIndex, CHAR *output_string);`
- `INT wifi_getApSecurityModeEnabled(INT apIndex, CHAR *output);`
- `INT wifi_getApSecurityKeyPassphrase(INT apIndex, CHAR *output_string);`
- `INT wifi_getApSecurityRadiusServer(INT apIndex, CHAR *IP_output, UINT *Port_output, CHAR *RadiusSecret_output);`
- `INT wifi_getSSIDEnable(INT ssidIndex, BOOL *output_bool);`
- `INT wifi_getApBridgeInfo(INT index, CHAR *bridgeName, CHAR *IP, CHAR *subnet);`
- `INT wifi_getApIsolationEnable(INT apIndex, BOOL *output);`
- `INT wifi_getApSsidAdvertisementEnable(INT apIndex, BOOL *output_bool);`
- `INT wifi_getSSIDName(INT apIndex, CHAR *output_string);`
- `INT wifi_getSSIDNameStatus(INT apIndex, CHAR *output_string);`
- `INT wifi_getBaseBSSID(INT ssidIndex, CHAR *output_string);`
- `INT wifi_getApMacAddressControlMode(INT apIndex, INT *output_filterMode);`
- `INT wifi_getApAclDevices(INT apIndex, CHAR *macArray, UINT buf_size);`
- `void wifi_newApAssociatedDevice_callback_register(wifi_newApAssociatedDevice_callback callback_proc);`
- `INT wifi_getApIndexFromName(CHAR *inputSsidString, INT *ouput_int);`
- `INT wifi_getRadioOperatingChannelBandwidth(INT radioIndex, CHAR *output_string);`

Setting Wi-Fi Information

The following settings are changed through wifi_hal using the APIs listed below:

- Radio Settings
  - Channel
  - Channel Width
- SSID Settings
  - Enable/Disable
  - ESSID
  - Encryption settings (mode, passphrase)
  - ACL configuration (mode, mac list)
  - CSA Deauth feature

    - AP ScanFilter feature (if required)

The following wifi_hal APIs are used to set or control various Wi-Fi parameters:

- `INT wifi_setSSIDEnable(INT ssidIndex, BOOL enable);`
- `INT wifi_setSSIDName(INT apIndex, CHAR *ssid_string);`
- `INT wifi_pushSSID(INT apIndex, CHAR *ssid);`
- `INT wifi_pushRadioChannel2(INT radioIndex, UINT channel, UINT channel_width_MHz, UINT csa_beacon_count);`
- `INT wifi_setApSecurityModeEnabled(INT apIndex, CHAR *encMode);`
- `INT wifi_setApSecurityKeyPassphrase(INT apIndex, CHAR *passPhrase);`
- `INT wifi_setApIsolationEnable(INT apIndex, BOOL enable);`
- `INT wifi_setRadioOperatingChannelBandwidth(INT radioIndex, CHAR *bandwidth);`
- `INT wifi_setApMacAddressControlMode(INT apIndex, INT filterMode);`
- `INT wifi_delApAclDevices(INT apIndex);`
- `INT wifi_addApAclDevice(INT apIndex, CHAR *DeviceMacAddress);`
- `INT wifi_applySSIDSettings(INT ssidIndex);`
- `INT wifi_setApSsidAdvertisementEnable(INT apIndex, BOOL enable)`

Wi-Fi Statistics

In a multi-hop Plume Cloud managed system, monitoring and optimizing of the topology based on current wireless conditions is mandatory. To be able to predict conditions and react on them, Plume Cloud system needs enhancements to the standard subset of the wireless parameters, which are divided per needed functionality.

Plume will manage all further SoC communication for statistics enhancements, and if their availability needs to be provided in the form of business rules through TR69, then additional work will need to be done between the AP Vendor and Plume. There are no direct requirements for the AP vendor in active mode except for turning off **ACS and scan**.

The following statistics are queried using the wifi_hal APIs listed below:

- wifi_neighbor_ap2_t
    - ap_SSID
    - ap_BSSID
    - ap_Mode
    - ap_Channel
    - ap_SignalStrength
    - ap_OperatingChannelBandwidth
- wifi_ssidTrafficStats2_t
    - ssid_BytesSend
- wifi_channelStats_t
    - ch_number
    - ch_utilization_total
    - ch_utilization_busy_tx

    - ch_utilization_busy_rx
    - ch_utilization_busy_self
    - ch_utilization_busy_ext
- wifi_associated_dev2_t
    - cli_MACAddress
    - cli_RSSI
- wifi_associated_dev_stats_t
    - cli_tx_bytes
    - cli_rx_bytes
    - cli_tx_frames
    - cli_rx_frames
    - cli_tx_rate
    - cli_rx_rate
    - cli_tx_retries
- wifi_associated_dev_rate_info_rx_stats_t
    - mcs
    - nss
    - bw
    - bytes
    - mpdus
    - ppdus
    - msdus
    - retries
    - rssi_combined
    - rssi_array
- wifi_associated_dev_rate_info_tx_stats_t
    - mcs
    - nss
    - bw
    - bytes
    - mpdus
    - ppdus
    - msdus
    - attempts

The following wifi_hal APIs are used for fetching Wi-Fi statistics:

- `INT wifi_setRadioStatsEnable(INT radioIndex, BOOL enable);`
- `INT wifi_getApAssociatedDeviceStats(INT apIndex, mac_address_t *clientMacAddress, wifi_associated_dev_stats_t *associated_dev_stats, ULLONG *handle);`
- `INT wifi_getApAssociatedDeviceRxStatsResult(INT radioIndex, mac_address_t *clientMacAddress, wifi_associated_dev_rate_info_rx_stats_t **stats_array, UINT *output_array_size, ULLONG *handle);`

- INT wifi_getApAssociatedDeviceTxStatsResult(INT radioIndex, mac_address_t *clientMacAddress, wifi_associated_dev_rate_info_tx_stats_t **stats_array, UINT *output_array_size, ULLONG *handle);
- INT wifi_getApAssociatedDeviceDiagnosticResult2(INT apIndex, wifi_associated_dev2_t **associated_dev_array, UINT *output_array_size);
- INT wifi_getApAssociatedDeviceDiagnosticResult3(INT apIndex, wifi_associated_dev3_t **associated_dev_array, UINT *output_array_size);
- INT wifi_getRadioChannelStats(INT radioIndex, wifi_channelStats_t *input_output_channelStats_array, INT array_size);
- INT wifi_startNeighborScan(INT apIndex, wifi_neighborScanMode_t scan_mode, INT dwell_time, UINT chan_num, UINT *chan_list);
- INT wifi_getNeighboringWiFiStatus(INT radioIndex, wifi_neighbor_ap2_t **neighbor_ap_array, UINT *output_array_size);
- INT wifi_getSSIDTrafficStats2(INT ssidIndex, wifi_ssidTrafficStats2_t *output_struct);

OpenSync Cloud Requirements

When connecting your gateway to the Plume cloud using the *OpenSync*, you will want to work with Plume on the following topics:

Certificate Authentication and Verification

You will need a certificate provided by Plume for your gateway. This certificate, signed by Plume, is used to authenticate the TLS connections from the *OpenSync* to the Plume cloud. We suggest having a certificate per gateway model number, and require that the certificate's private key be stored encrypted within the firmware image.

Model Information

The Plume cloud makes a lot of decisions based on the model number reported by the gateway. These decisions include supported features, Wi-Fi capabilities such as number of antennas and spatial streams, supported channels, transmit power, etc.

OpenSync Wi-Fi Statistics Certification

Plume is working on a statistics certification program, where the Wi-Fi statistics being reported by the gateway and Wi-Fi chip vendor are validated based on a predefined set of tests. These tests also allow a form of calibration, so that statistics reported by various vendors -- including those of the *OpenSync* enabled extenders -- can be analysed together by the Plume data pipeline and adaptive Wi-Fi system.

Plume Steering

The Plume cloud -- through the *OpenSync* -- provides band and client/AP steering. It will make decisions based on various data points, such as whether clients should be moved across bands on the same AP, or to another AP (gateway or extender). The steering algorithm is outside the scope of this document.

Requirements

The API's required for this feature are listed above in the *RDK-B wifi_hal Requirements* section. The foundation of this support are events provided by the lower-level driver, delivered to the callback registered using the *wifi_steering_eventRegister()* API call. Without these events, the steering feature will not function.

Per-Client Configuration

One of the unique features of this steering is that it contains per-client configuration. This includes the following values:

- Client MAC Address
- Probe High Watermark, Probe Low Watermark
    - Only probe requests with RSSI in this range should be answered
- Auth High Watermark, Auth Low Watermark
    - Only auth requests with RSSI in this range should be answered
- High RSSI Crossing Value, Low RSSI Crossing Value
    - Crossing events should be generated when the client's RSSI goes above or below these thresholds
- Auth Reject Reason
    - If value is > 0, then auth requests should be rejected with the given reason code when they are outside the RSSI range configured above, instead of being silently dropped.

Steering Events from Wi-Fi Driver

The following steering events are required from the Wi-Fi driver:

WIFI_STEERING_EVENT_PROBE_REQ

Receiving probe requests from an added steering client, along with details of that probe request, is a vital part of the steering system. For instance, signal strength of the received probe request frame can be used to estimate the connection quality between a non-associated client.

The steering API provides a mechanism to configure the RSSI thresholds, within which the probe requests should be answered. If the RSSI is outside of that range, then the probe request should be blocked, and it must be marked as such in the event.

This event should be delivered whenever a probe request is received from a client added using the proper wifi_hal_steering API call. The event must implement the following parameters:

- Client's MAC Address
- RSSI of the probe request frame
- Whether the probe request is a broadcast probe (i.e. NULL probe request, no ESSID provided)
- Whether the probe request was blocked by the driver due to steering configuration.

WIFI_STEERING_EVENT_CLIENT_CONNECT

This event should be delivered whenever an added steering client connects, and is used by the steering algorithm to determine whether steering succeeded or failed.

WIFI_STEERING_EVENT_CLIENT_DISCONNECT

This event should be delivered whenever an added steering client disconnects, and is used to reset the steering algorithm.

WIFI_STEERING_EVENT_CHAN_UTILIZATION

This event should be delivered at the pre-configured interval, and should provide the current channel utilization. This can be used by the algorithm when performing pre-association band steering.

WIFI_STEERING_EVENT_RSSI_XING

The steering algorithm may make decisions based on a connected client RSSI crossing some threshold. Currently there are two thresholds which are configured ***per client*** when they are added using the steering API (a high threshold and a low threshold).

When the client crosses above or below either of these values, this event should be generated with the following parameters:

- Client's MAC Address
- Client's current RSSI
- High XING status (unchanged, higher, lower)
- Low XING status (unchanged, higher, lower)

WIFI_STEERING_EVENT_RSSI

Depending on how the RSSI crossing event system works within the Wi-Fi driver, it may be possible to have false triggers. The steering API provides a mechanism to perform an instant RSSI measurement of a connected client. While not required, it is strongly recommended.

One example of how this works is to transmit a pre-configured number of NULL frames to the client, and to average the RSSI of the NULL frame ACKs from that client. The averaged value is then provided by this event.

If this is not implemented, then the supplied client Measure API should return `-ENOSYS`.

WIFI_STEERING_EVENT_AUTH_FAIL

The algorithm makes decisions on if/when AUTH frames are blocked or rejected based on the steering configuration provided when a client is added. This configuration includes an RSSI range, and when the RSSI of the AUTH frame is outside of that range, it should either be silently dropped, or rejected with a given reason code as defined by the steering configuration. This action, whichever taken, must be included in the event.

This event should be delivered whenever a Wi-Fi AUTH request failure occurs, and should provide the following parameters:

- Client's MAC Address
- RSSI of the AUTH frame
- Blocked status: Whether the AUTH request was blocked due to steering configuration
- Rejected status: Whether the AUTH request was rejected or silently dropped
- Reason: Reason code, if the AUTH request was rejected.

Steering API

The following wifi_hal APIs are used for steering:

- `BOOL wifi_steering_supported(void);`
- `INT wifi_steering_setGroup(UINT steeringgroupIndex, wifi_steering_apConfig_t *cfg_2, wifi_steering_apConfig_t *cfg_5);`
- `INT wifi_steering_eventRegister(wifi_steering_eventCB_t event_cb);`
- `INT wifi_steering_eventUnregister(void);`
- `INT wifi_steering_clientSet(UINT steeringgroupIndex, INT apIndex, mac_address_t client_mac, wifi_steering_clientConfig_t *config);`
- `INT wifi_steering_clientRemove(UINT steeringgroupIndex, INT apIndex, mac_address_t client_mac);`
- `INT wifi_steering_clientMeasure(UINT steeringgroupIndex, INT apIndex, mac_address_t client_mac);`

- INT wifi_steering_clientDisconnect(UINT steeringgroupIndex, INT apIndex, mac_address_t client_mac, wifi_disconnectType_t type, UINT reason);
- INT wifi_setRMBeaconRequest(UINT apIndex, CHAR *peer, wifi_BeaconRequest_t *in_request, UCHAR *out_DialogToken);
- INT wifi_setBTMRequest(UINT apIndex, CHAR *peerMac, wifi_BTMRequest_t *request);
- INT wifi_getBSSTransitionActivation(UINT apIndex, BOOL *activate);
- INT wifi_setBSSTransitionActivation(UINT apIndex, BOOL activate);
- INT wifi_getNeighborReportActivation(UINT apIndex, BOOL *activate);
- INT wifi_setNeighborReportActivation(UINT apIndex, BOOL activate);
- INT wifi_RMBeaconRequestCallbackRegister(UINT apIndex, wifi_RMBeaconReport_callback beaconReportCallback);
- INT wifi_BTMQueryRequest_callback_register(UINT apIndex, wifi_BTMQueryRequest_callback btmQueryCallback, wifi_BTMResponse_callback btmResponseCallback);
- INT wifi_RMBeaconRequestCallbackUnregister(UINT apIndex, wifi_RMBeaconReport_callback beaconReportCallback);

Multi-AP DFS

The Plume cloud -- through *OpenSync* -- provides and orchestrates DFS across all devices at a location. In order to minimize DFS CAC timer effect on user experience, the Cloud needs to be aware of the device's internal DFS states:

- **CAC** (Channel Availability Check): The period of time in which AP will monitor for presence of radar signals.
- **NOP** (Non-Occupancy Period): The period of time in which a radar detected channel will become unusable channel (or unavailable channel).
- **ISM**: It is a process which will continuously monitor the operating channel for detecting the presence of radar signals.

The Cloud looks at the DFS states and derives topologies that ensure uninterruptible service to the user. At the beginning, the Cloud uses a "temporary topology" while CAC on the selected channels is being performed. After CACs are complete, the Cloud configures the "final topology", which is a result of the Optimization process (environment changes and RADAR events re-trigger optimization).

The DFS algorithms are outside the scope of this document. Plume DFS requirements specify the parameters that are needed for maintaining CAC and NOP stats in the Non-Occupancy List (NOL). NOL is a list of channels that are unavailable at any given point in time on any specific device.

DFS States

DFS states provide the internal status of a DFS channel:

- **ALLOWED** - channel is available to use
- **CAC_STARTED** - CAC has been started and is in progress
- **CAC_COMPLETED** - CAC has completed successfully, AP VAPs now UP
- **NOP_STARTED** - RADAR event received and NOP started
- **NOP_FINISHED (CAC_READY)** - NOP finished and the channel can be reused

NOTE: Nice to have also CAC elapsed time or NOP left time.

```
Requirement: An API for getting the DFS channel list with states:
channels CHAR[ ][ ]
                │  │
                │  │
                │  └──────────── state:"ALLOWED"
                │                     :"CAC_STARTED"
                │                     :"CAC_COMPLETED"
                │                     :"NOP_STARTED"
                │                     :"NOP_FINISHED"
                │
                └─────  chan_idx
```

DFS Events

The Cloud Optimization must be aware of the channel changes events to prevent misconfiguration and ensure DFS standard compliance:

- **RADAR** - A radar event notification, containing the channel number
- **CHANNELS_CHANGED** - An event notifying a channel change as result of CAC/NOP changes

```
Requirement: An API for subscribing to RADAR and CHANNEL CHANGE notification/
events need to be provided. The event should contain:

- channel number
- state

NOTE: OpenSync re-reads the whole radio state upon the channel changed event
- target_radio_state_get()
```

Channel Changes

In order to ensure minimum impact on user experience, the following changes are required (some of them might be already a part of SoC vendor's BSP):

- Driver should support that NOP_FINISHED state events in case of channel changes from DFS -> non-DFS are received (even after PRE-CAC)
- In case of a RADAR event, the CSA to a non-DFS channel needs to be used (if a non-DFS is not available, just switch to a random one)

Example of events:

```
wifi0 Custom driver event:RADAR: CAC started for channel  64 (5320 MHz)
wifi0 Custom driver event:RADAR: CAC completed for channel  64 (5320 MHz)
wifi0 Custom driver event:RADAR: radar found on channel 108 (5540 MHz)
wifi0 Custom driver event:RADAR: non-occupancy period expired for channel 108 (5540 MHz)
```

Wifi HAL DFS API (available since 2.15):

The following APIs are wifi_hal for DFS:

- `INT wifi_getRadioDfsSupport(INT radioIndex, BOOL *output_bool);`
- `INT wifi_getRadioDfsEnable(INT radioIndex, BOOL *output_bool);`
- `INT wifi_setRadioDfsEnable(INT radioIndex, BOOL enabled);`
- `INT wifi_chan_eventRegister(wifi_chan_eventCB_t event_cb);`
- ```
  typedef enum {
      WIFI_EVENT_CHANNELS_CHANGED,
      WIFI_EVENT_DFS_RADAR_DETECTED
  } wifi_chan_eventType_t;
  ```
- `typedef void (*wifi_chan_eventCB_t)(UINT radioIndex, wifi_chan_eventType_t event, UCHAR channel);`
- ```
  typedef enum {
     CHAN_STATE_AVAILABLE = 1,
     CHAN_STATE_DFS_NOP_FINISHED,
     CHAN_STATE_DFS_NOP_START,
     CHAN_STATE_DFS_CAC_START,
     CHAN_STATE_DFS_CAC_COMPLETED
  } wifi_channelState_t;
  ```
- ```
  typedef struct _wifi_channelMap_t {
     INT ch_number;
     wifi_channelState_t ch_state;
  } wifi_channelMap_t;
  ```
- `INT wifi_getRadioChannels(INT radioIndex, wifi_channelMap_t *output_map, INT output_map_size)`
  ```

Multi-PSK

Plume's Guest Access with Plume *HomePass*™ takes a novel groundbreaking approach to managing access to the home Wi-Fi SSID. Instead of creating a separate "guest SSID", a single SSID is used at each location, a number of access **zones** are created to manage access privileges of connecting devices.

Each access zone is accessible using a unique set of keys (Wi-Fi passwords), any of which can be used to access the SSID. There is no technical upper limit on the number of keys that can be assigned to each zone - but to keep this manageable, the PRD sets this limit to 10. For now, we treat all passwords in an access zone the same, but in the future we may allow passwords to have different ACLs.

The key used to access the SSID determines the access zone for the connecting device. Specifically, a device is automatically a part of the access zone for which it is connected, and if a device has been given multiple passwords then its zone is determined by which password it most recently used to connect.

> **Requirement:** An API for configuring MultiPSK per AP interface (VIF) must be provided. The API must contain:
> - interface name
> - key_id
> - psk

For *OpenSync* to know which rules need to be applied for particular client, the event containing the password upon association needs to be provided:

> **Requirement:** An API for subscribing to 'client reconnect' events is needed to get the PSK used during association. The events should contain:
> - interface name
> - client's MAC address
> - key_id
> - associated state

wifi_hal proposal for Multi-PSK

The following APIs are wifi_hal proposal for Multi-PSK:

```
typedef struct _wifi_associated_dev4
{
       ...
       CHAR  cli_keyId[64];       // The key ID of associated device.
} wifi_associated_dev4_t;

typedef INT (* wifi_newApAssociatedDevice_callback)(
            INT apIndex, wifi_associated_dev4_t *associated_dev);

void wifi_newApAssociatedDevice_callback_register(
            wifi_newApAssociatedDevice_callback callback_proc);

INT wifi_getApAssociatedDeviceDiagnosticResult4(
            INT apIndex,
            wifi_associated_dev4_t **associated_dev_array,
            UINT *output_array_size);

typedef INT (* wifi_apDisassociatedDevice_callback)(
            INT apIndex, char *MAC, INT event_type);

void wifi_apDisassociatedDevice_callback_register(
            wifi_apDisassociatedDevice_callback callback_proc);
```

New APIs:

```
/* wifi_addKeyMultiPsk() function */
/**
* @brief Add key to AP.
*
* @param apIndex  access point index
* @param keyId  key index (max length: 64)
* @param psk  password (max length: 64)
* @param mac  MAC of the client (length: 32)
*
* @return The status of the operation
* @retval RETURN_OK if successful
* @retval RETURN_ERR if any error is detected
*
* @sideeffect None
*/
INT wifi_addKeyMultiPsk(INT apIndex, const CHAR *keyId, const CHAR *psk, const CHAR *mac);

/* wifi_removeKeyMultiPsk() function */
/**
* @brief Remove key from AP.
*
* @description Remove key from AP. Clients that are connected with the key
* being removed should be kicked off from the network.
*
* @param apIndex  access point index
```

```
* @param keyId    key index (max length: 64)
*
* @return The status of the operation
* @retval RETURN_OK if successful
* @retval RETURN_ERR if any error is detected
*
* @sideeffect None
*/
INT wifi_removeKeyMultiPsk(INT apIndex, const CHAR *keyId);

/* wifi_isApMultiPskSupported() function */
/**
* @brief Checks if access point supports multiPsk.
*
* @param apIndex  access point index
*
* @return The status of AP
* @retval TRUE AP supports multiPsk
* @retval FALSE AP does not support multiPsk
*
* @sideeffect None
*/
BOOL wifi_isApMultiPskSupported(INT apIndex);

/* wifi_newApAssociatedDeviceMultiPsk_callback_register() function */
/**
* @brief Callback registration function.
*
* @param[in] callback_proc  wifi_newApAssociatedDeviceMultiPsk_callback callback function
*
* @return The status of the operation
* @retval RETURN_OK if successful
* @retval RETURN_ERR if any error is detected
*
* @execution Synchronous
* @sideeffect None
*
* @note This function must not suspend and must not invoke any blocking system
* calls. It should probably just send a message to a driver event handler task.
*/
void wifi_newApAssociatedDeviceMultiPsk_callback_register(
            wifi_newApAssociatedDeviceMultiPsk_callback callback_proc);

/* wifi_overrideAllKeysMultiPsk() function */
/**
* @description Function sets the whole current keys config. It should check
* if there were any changes against previous keys and apply them. If any key
* is removed or key ID is changed, devices that are associated with this key
* should be kicked off from the network.
*
* @note Function must free keys memory.
*
* @param apIndex  access point index
* @param keys     all keys that devices can associate with AP
* @param array_size  number of keys
*
* @return The status of the operation
* @retval RETURN_OK if successful
* @retval RETURN_ERR if any error is detected
*
* @sideeffect None
*/
INT wifi_overrideAllKeysMultiPsk(INT apIndex, wifi_key_multi_psk_t **keys, INT array_size);
```

Operational Requirements

OpenSync is currently running solely on the Linux operating system, where the recommended version to get all the features is **Kernel 3.3 or higher**.

Purpose	Library	Minimum Version
toolchain	[libc.so.0]	
	[libdl.so.0]	
	[libgcc_s.so.1]	
	[libpthread.so.0]	
	[librt.so.0]	
	[libm.so.0]	
system	[libssl.so.1.0.0]	openssl-1.0.2k
	[libcares.so.2]	c-ares-1.10.0
	[libz.so.1.2.8]	zlib-1.2.8
	[libncurses.so.5.9]	ncurses-5.9
applications	[libjansson.so.4]	jansson-2.7
	[libmosquitto.so.1]	mosquitto-1.4.8
	[libprotobuf-c.so.1]	protobuf-c-1.1.1
	[libev.so.4]	libev-4.24
openvswitch	[libovsdb.so.1.0.0]	Openvswitch-2.8.7
	[libopenvswitch.so.1.0.0]	

Versions of required OVS libs and tools depend on the version of the Linux kernel running on the GW. Please check reference table in OVS documentation:

- http://docs.openvswitch.org/en/latest/faq/releases/