User Guide

Wi-Fi Module REST API Specification

For Black Box

Version: 3.0

Date Released: June 07, 2016

Document Number: AY006UWF3-3

Ayla Networks

Referenced Documents

Ayla Networks does not supply all documents that are referenced in this document with the equipment. Ayla Networks reserves the right to decide which documents are supplied with products and services.

Contact Information

Ayla Networks TECHNICAL SUPPORT and SALES

Contact Technical Support: https://support.aylanetworks.com
or via email at support@aylanetworks.com

Contact Sales: https://www.aylanetworks.com/company/contact-us

Ayla Networks REGIONAL OFFICES

GREATER CHINA
Shenzhen
Room 310-311
City University of Hong Kong
Research Institute Building
No. 8 Yuexing 1st Road
High-Tech Industrial Park
Nanshan District
Shenzhen, China
Phone: 0755-86581520

HEADQUARTERS
Silicon Valley
4250 Burton Drive, Suite 100
Santa Clara, CA 95054
United States
Phone: +1 408 830 9844
Fax: +1 408 716 2621

JAPAN
Wise Next Shin
Yokohama, 2-5-14
Shnyokohama, Kohokuku
Yokohama-shi, Kanagawa-ken
Yokohoma, 222-0033 Japan

EUROPE
London
30 Great Guildford St
London SE1 0HS
United Kingdom

TAIWAN
Taipei
5F No. 250 Sec. 1
Neihu Road, Neihu District
Taipei 11493, Taiwan

For a Complete Contact List of Our Offices in the US, China, Europe, Taiwan, and Japan:
https://www.aylanetworks.com/company/contact-us

Table of Contents

1 Introduction

This document provides the REST API specification for the Ayla Wi-Fi module.

1.1 Audience

This document is for engineers and assumes experience with programming.

1.2 Related Documentation

N/A

2 Access Point Mode

This document describes the JSON interface that is used to access configuration and properties for the Ayla module while it is in Access Point (AP) mode. This interface is provided by the module's internal web service.

2.1 AP Mode Configuration

The module's SSID is configured as part of its factory settings. To use AP mode, connect to the module's Wi-Fi network, use the IP address 192.168.0.1, or whatever IP address is configured to be self-assigned by the module in AP mode. Any other host name redirects to that address.

NOTE	In this document, the JSON has white space added for clarity. The module may or may not emit white space.

2.1.1 Common Token Definitions

Below are the definitions of several fields that are commonly used in the JSON-encoded data returned or accepted by the operations described throughout the document.

- **mtime** - module's internal monotonically increasing time in milliseconds starting at either startup or power-on, depending on the module's capabilities.
- **time** - UTC time in seconds since 1970. On current modules it survives across reboots as long as power is maintained. On startup, the module firmware sets it to January 1, 2012, 00:00 (subject to change yearly) if it is not already past that time.

2.2 Device Operations

2.2.1 Get Module Status

```
$ curl http://192.168.0.1/status.json
{
  "dsn": "AC000W000000123",
  "device_service": "ads-dev.aylanetworks.com",
  "mac": "12:34:56:78:90:12",    /* optional field */
  "last_connect_mtime": 12222,    /* TBD change to time */
  "last_connect_time": 1363798365,
  "mtime": 4567,
  "model": "AY001MUS1",    /* optional field */
  "version": "0.10.1",
  "features": [ "ap-sta",... ],     /* optional field */
  "api_version": "1.0",
```

```
 "build": "bc 0.10.1 01/19/12 17:37:50 ID jre/log/f69f8c7+",
}
```

This gets the module ID information and device service configuration.

- "dsn" - device serial number, which is the key identifier used by the device service.
- "device_service" - host name used to contact the device service. This is configured in mfg and setup mode and generally not changeable.
- "mac" - module's assigned MAC address. This field is optional, to get it use wifi_status.json.
- "last_connect_mtime" - the **mtime** at which the module last successfully connected to the service, or 0 if no connection has been made.
- "last_connect_time" - the **time** at which the module last successfully connected to the service, or 0 if no connection has been made.
- "model" - Ayla part number for the module, if applicable.
- "features" – (optional) The value contains an array of intrinsically supported feature names. Defined values include:
  - "ap-sta" - simultaneous AP and STA mode
  - "wps" - Wi-Fi Protected Setup
  - "rsa-ke" - RSA key exchange is required for the initial wifi_connect.json call
- "api_version" - version of this set of JSON interfaces. It can be of the form "x.y" where x is the major version, y is the minor version. If the major version changes, some of the interfaces may no longer be supported. If only the minor version changes, some new interfaces may be available, but any changes are compatible with older applications. Unless otherwise noted, all these interfaces are available in all API versions. If "api_version" is not present, it should be assumed to be "1.0".

2.2.2 Get Module Registration Token

To get the registration code needed to verify connectivity to the device when registering the device with the device service, issue the following GET request:

```
$ curl http://192.168.0.1/regtoken.json
{
  "regtoken": "abcde99",
  "registered": 1,    /* 1 or 0, in 1.6 or later */
  "registration-type": "Same-LAN"    /* after 1.6 */
  "host_symname":"Sprinkler Controller"    /* after 1.6 */
}
```

The regtoken may be null if the module is not able to contact the device service, or if the registration type is "Display".

The registration-type may be an empty string if the module is not able to contact the device service.

2.2.3 Get Module Time

```
$ curl http://192.168.0.1/time.json
{
  "time": 1398449735,
  "mtime": 10000,
  "set_at_mtime": 5000,
  "clksrc": 0,
  "localtime": "2014-04-25T11:15:35",
  "daylight_active": 0,
  "daylight_change": 1414918800,
}
```

Parameter descriptions are:

- "time" - UTC time in seconds since midnight of January 1, 1970.
- "mtime" - module time in milliseconds since last reset.
- "set_at_mtime" - mtime at which the time was last set, or 0 if unknown.
- "clksrc" - internal value indicating from where the clock setting came. This should be used for diagnostic purposes only, and is not to be relied upon.
- "localtime" - ASCII string decoding the local time. The format of this string is subject to change.
- "daylight_active" - 1 if a one-hour daylight savings time adjustment is or was being added until the daylight_change time, and 0 otherwise. After the daylight_change time, the one-hour daylight savings adjustment is applied only if daylight_active is 0. See the table below for details.
- "daylight_change"- UTC time at which the daylight_active flag should be considered inverted.

Table 1 Daylight Savings Explanation

daylight_valid	daylight_active	time	DST adjustment
0	n/a	n/a	None
1	0	before daylight_change	None
1	0	after daylight_change	Add 1 hour
1	1	before daylight_change	Add 1 hour
1	1	after daylight_change	None

2.2.4 Time Conversion Example

To reliably convert mtimes in messages, such as, the wifi_scan_results.json response, follow this procedure:

1. Get the module time, which has the **time** and corresponding **mtime** value.
2. Let **epoch** = **time** - **mtime** / 1000. This is the time the module last started or some other arbitrary recent time when **mtime** was 0.
3. If **set_at_mtime** is 0, use your local time instead of the **time** returned from the module.
4. Let **result_secs** = **epoch** + **mtime_from_message** / 1000. This is the time since 1970 in seconds. Allow for it to be more than 32-bits wide and unsigned.
5. Let **result_ms** = **epoch** * 1000 + **mtime_from_message**. This is the time since 1970 in milliseconds. It is more than 32-bits wide and unsigned.

2.2.5 Set Module Time

```
$ curl -X PUT -d '{"time":87654321}' -H "Content-Type: application/json"
http://192.168.0.1/time.json
```

This should always succeed, returning status 204 (No Content). If the data is incorrectly formatted, returns status "400 Bad Request".

2.3 Wi-Fi operations

2.3.1 Get Wi-Fi Status

```
$ curl http://192.168.0.1/wifi_status.json
{"wifi_status": {
  "connect_history":  [ {
  "ssid_info": "e1",
  "ssid_len": 7,
  "bssid": "1234",
  "error": 4,
  "msg": "join failed: bad key",
  "mtime": 2000,
  "last": 0,
  "ip_addr": "192.168.0.1",
  "netmask": "255.255.0.0",
  "default_route": "192.168.0.1",
  "dns_servers": [ "192.168.0.1", "172.16.1.30" ]
 },
 {
  "ssid_info": "r7",
```

```
    "ssid_len": 5,
    "error": 0,
    "bssid": "578c",
    "msg": null,
    "mtime": 5000,
    "last":0,
    "ip_addr": "192.168.0.1",
    "netmask": "255.255.0.0",
    "default_route": "192.168.0.1",
    "dns_servers": [ "192.168.0.1", "172.16.1.30" ]
   },
   {
    "ssid_info": "b2",
    "ssid_len": 7,
  "error": 3,
    "bssid": "a12c",
    "msg": "join timed out",
    "mtime": 3600000,"last":0,
    "ip_addr": "192.168.0.1",
    "netmask": "255.255.0.0",
    "default_route": "192.168.0.1",
    "dns_servers": [ "192.168.0.1", "172.16.1.30" ]
   }],
    "dsn": "AC000W000000123",
    "device_service": "ads-dev.aylanetworks.com",
    "mac": "12:34:56:78:90:12",
    "mtime": 12345,    /* after 1.6 */
    "host_symname":"Sprinkler Controller",    /* after 1.6 */
    "connected_ssid":"Ayla-dev",    /* after 1.6 */
    "ant": 0,
    "wps": "none",    /* WPS status */
    "rssi": -40,
    "bars": 5
    }
  }
```


The Connect_history is null if there were no connection attempts since reset. Otherwise, it is an array containing the recent connection results. At least the last three attempts since the module was reset are shown, with the most recent one first.

The ssid_info is the first and last characters of the SSID used. This is to partially anonymize the data. The ssid_len is the number of bytes in the SSID. The "bssid" value is the last 4 hex digits of the BSSID. The IP address data may be anonymized at a later date. The "last" field records when module makes the last attempt to connect to a particular AP.

Table 2. Error Codes and Descriptions

Error codes	Description
0	No error
1	Resource problem, out of memory or buffers, perhaps temporary.
2	Connection timed out.
3	Invalid key.
4	SSID not found.
5	Not authenticated via 802.11 or failed to associate with the AP.
6	Incorrect key
7	Failed to get IP address from DHCP
8	Failed to get default gateway from DHCP
9	Failed to get DNS server from DHCP
10	Disconnected by AP
11	Signal lost from AP (beacon miss)
12	Device service host lookup failed
13	Device service GET was redirected
14	Device service connection timed out
15	No empty Wi-Fi profile slots
16	The security method used by the AP is not supported.
17	The network type, for example, is not supported.
18	The server responded in an incompatible way. The AP may be a Wi-Fi hotspot.
19	Device service authentication failed
20	Connection attempt is still in progress

NOTE rssi, bars, and ant are optional fields and may not be provided by all versions.

2.3.2 Start Wi-Fi Scan

To start a new Wi-Fi scan, do the following:

```
$ curl -X POST -H "Content-Type: application/json"
http://192.168.0.1/wifi_scan.json
```

This returns status 204 (No Content) and starts a scan.

2.3.3 Get Wi-Fi Scan Results

This operation gets the most recent scan results.

```
$ curl http://192.168.0.1/wifi_scan_results.json


{ "wifi_scan": {
  "mtime": 13000,
  "results": [
  {
    "ssid": "example1" ,
    "type": "AP",
    "chan": 11,
    "signal": -52,
    "bars": 3,
    "security": "WPA2 Personal Mixed",
    "bssid": "60e956000001"
  },
  {
    "ssid": "example2",
    "type": "Ad hoc",
    "chan": 3,
    "signal": -65,
    "bars": 2,
    "security": "None",
    "bssid": "60e965000002"
   }
  ]
 }
}
```

The scan results are sorted by descending signal strength.

mtime is the module time in milliseconds at which the last scan was started.

The SSIDs are URI-encoded, so that arbitrary 32-byte hex strings are possible.

The security strings are the types returned by Wiced and are different than the security values used in the profiles. For now the security values used are one of these: "None", "WEP", "WPA", "WPA2 Personal AES", "WPA2 Personal Mixed", or "Unknown" (for example,, for WPS).

```
$ curl http://192.168.0.1/wifi_scan_results.json?nossid
```

If 'nossid' argument is provided, scan results are reported without SSIDs.

2.3.4 Connect to Network

A connect request looks like this:

```
$ curl -X POST
'<host>/wifi_connect.json?ssid=example1&key=top%20secret%20key&setup_token=
01a932'
```

If the setup_token is provided, it should be hex digits or other random string of up to eight characters long that is passed to the device service and can be checked to "prove" the module has connected as requested.

To connect to a hidden network, the "hidden=1" parameter is added. The request looks like this:

```
$ curl -X POST '<host>/wifi_connect.json?ssid=example1&setup_token=1357ac&\
key=top%20secret%20key&\
hidden=1'
```

Response:

The response may be empty or the connection may merely drop.

If the HTTP response is received, it'll either have status 204 (No Content), meaning the connection will be attempted, or an error response of with a body indicating the error:

```
{ "error": 3, "msg": "Invalid Key" }
```

A successful response means the connection is initiated. If the module was in AP mode, it drops AP mode after some time, if it succeeds in connecting to the service. If that is unsuccessful, it stays in AP mode, and you can obtain the error reason via the status URL.

If the connection allows the module to complete a request to the device service, a profile is created and the running configuration is saved to the startup configuration.

The app can pass the location of the device using a "location" flag. For example:

```
curl -X POST
'<host>/wifi_connect.json?ssid=example1&key=top%20secret%20key&setup_token=
01a932&location=37.123456,-127.123456'
```

Optionally, after bc-1.6, the BSSID may be specified instead of the SSID. Example:

```
$ curl -X POST
'<host>/wifi_connect.json?bssid=020304050607&key=top%20secret%20key&setup_t
oken=01a932'     # new in 1.6
```

2.3.5 Connect with WPS

Ayla supports Wi-Fi Protected Setup (WPS) with Press Button Configuration in bc releases 1.6 or later. App can do a virtual button press when module is in Access Point mode.

Button press request looks like this:

```
$ curl -X POST 'http://192.168.0.1/wps_pbc.json'
```

Response:

This returns status 204 (No Content) and starts a WPS scan. Report of the WPS status is in wifi_status.json.

2.3.6 Get Wi-Fi Profiles

```
$ curl http://192.168.0.1/wifi_profiles.json
{ "wifi_profiles": [
    { "ssid": "example1",
      "security": "WPA2 Personal"
    },
    { "ssid": "example2",
      "security": "WEP"
    }
  ]
}
```

The security value is one of these: "none", "WEP", "WPA", or "WPA2 Personal".

2.3.7 Delete Profile / Disconnect

```
curl -X DELETE  'http://192.168.0.1/wifi_profile.json?ssid=example2'
```

Response:

204 No Content, on success

404 Not Found

2.3.8 Disconnect Access Point mode

```
curl -X PUT 'http://192.168.0.1/wifi_stop_ap.json'
```

Response:

On success, an HTTP status of "204 No Content" is returned. Otherwise, an HTTP status of "403 Forbidden" is returned.

AP mode disconnect command is only accepted when module is still in AP mode, and successfully connected to service in Station Infrastructure Mode (STA) mode, that is, both STA mode and AP mode are active at the same time. This command turns off the AP mode, leaving STA mode active only.

3 LAN Mode

This section describes the direct communication between a Mobile Application and the Module. This is referred to as WLAN Local Mode, or LAN mode. LAN Mode is the default mode for transferring events between a Mobile Application and the Module. Implementation is available to Mobile Applications via the Ayla Mobile Libraries: aAML (Android), and iAML (iOS). Other client implementations should conform to this specification.

LAN mode communication fulfills the following goals:

- Reduce latency of event generation and consumption, without compromising security or power management.
- Enable Module to Mobile Application communication when the Device Service is not reachable.
- Module supports up to two simultaneous LAN mode mobile applications.
- Configurability of important attributes on the Device Service with OEM scope

3.1 Module Discovery

The Mobile Application saves the Module MAC address to NVRAM during the setup. In addition, it saves the IP Address from the last time the Module was connected to the device service. The Module lan_ip is saved to NVRAM on the successful conclusions of device setup and updated, if necessary, on completion of device registration. In addition, the lan_ip is checked and updated as necessary during every GET apivX/device(s) Device Service call.

The mobile app uses the last known IP address saved in NVRAM to register with the Module for local event notifications. If a connection cannot be established, the mobile app uses multicast DNS to discover the Module IP Address. For Android, the Mobile Application implements standard mDNS. For iOS, the Mobile Application implements Bonjour discovery, which is based on mDNS.

Because multicast address is not retried on a WLAN, the Mobile Application retries the discovery. The number of retries is based on the xAML SystemUtils.WifiRetries value. The discovery method will not return until the IP address is known, or all retries have been exhausted. A public progress indicator reflects the retry count. Finally, if the Module has still not been discovered, the Mobile Application uses the lan_ip returned from the Device Service.

3.2 Security

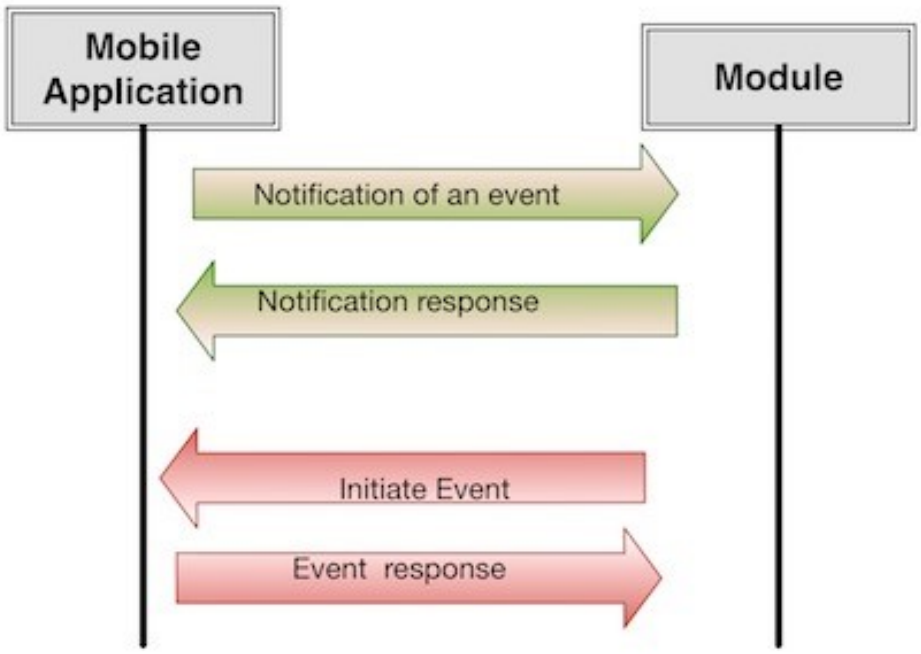
This section describes the security mechanisms and top-level flows put in place to securely exchange information between the Module and the Mobile App. These mechanisms include a shared secret, authentication signing, and data encryption methods.

The goal is to create a secure communication session between the Mobile App server and Module client. The Module server and Mobile Application client communication session will remain open/insecure.

```
Open notification path: Mobile HTTP client -> Module HTTP server
Secure data session: Mobile App HTTP server <- Module HTTP client
```

The arrows above indicate which entity initiates the exchange (with one exception). The overall flow is designed to emulate the client/server flow between the Module and the Device Service. The open notification path is designed to emulate the "in the clear" ANS, and the secure session is used to exchange confidential information.

Figure 1: Event Activities



3.2.1 Shared Secret

There is one shared secret per Module. To support it, the Ayla Device Service provides an API with three attribute fields for each device:lanip_key, lanip_key_id, and lanip_key_expiry. They contain a shared secret between the Module and the mobile app along with an associated sequence count, and an expiry time, respectively.

The values are generated by the device service during registration and updated every six months by default. However, the shared secret refresh interval is configurable during OEM configuration. The lanip_key is 32 random characters which may be alphanumeric or '='. The lanip_key_id is a 16 bit unsigned integer so as to avoid rollover issues. The expiry is an integer value time in seconds.

When the lanip_key is first created or regenerated by the Device Service, the sequence count is incremented and the Module is notified via the standard Ayla Notification Service and updated via the Notification Commands processes. When the Module successfully updates to the new lanip_key the Device Service makes it, and the associated lanip_key_id, available for downloading to the Mobile Application.

The mobile app gets the lanip_key and lanip_key_id from the device service via the GET apiv1/devices.[json|xml] and/or GET apiv1/devices/<key>.[json|xml] calls. These values are saved to NVRAM so they are available when the Device Service is unreachable. Usage of the secret and the sequence number are described below for the Mobile Application and for the Module.

3.2.2 Message Integrity

The lanip_key is used along with 16 random alphanumeric characters to create a secure session key in accordance with RFC 5246, Section 5 (TLS PRF with SHA256 as HMAC. The SHA256 hash of the session key is used for signing each packet. Details can be found below.

3.2.3 Encryption

The lanip_key is also used, along with 16 random alphanumeric characters to create a session key for encryption. The process to derive the key is similar to the signing key, but using different random data. This session key is used with an AES256 CBC based hash with an initialization vector to encrypt the HTTP message/payload. Details, including message formatting follow.

3.2.4 Registration/Notification/Session Extension

The module is notified that of the Mobile Application's desire for communication by the Application sending the Module a local registration (local_reg) request. This is a notification to establish a secure session between the two entities, if one is not already established.

If a secure session hasn't been established, the Mobile App client registers with the module:

```
$ curl -X POST -d
'{"local_reg":{"ip":"192.168.0.2","port":10275,"uri":"local_lan","notify":1
}}' -H "Content-Type: application/json"   http://192.168.0.1/local_reg.json
```

Status Codes:

202: Accepted - The Module initiates a key exchange (if needed) followed by a GET

400: Bad Request (Json Parse failure in request)

404: Not Found - LAN Mode is not enabled for this module

412: Precondition Failed - Module does not have lanip key

403: Forbidden - lan_ip on a different network

503: Service Unavailable - Insufficient resources or maximum number of sessions exceeded

In order to extend an existing secure connection, the Mobile App client sends a local registration (local_reg) request using PUT:

```
$ curl -X PUT -d
'{"local_reg":{"ip":"192.168.0.2","port":10275,"uri":"local_lan","notify":<
```

```
0|1>}}' -H "Content-Type: application/json"
http://192.168.0.1/local_reg.json
```

Status Codes:

202: Accepted - The Module has acknowledged the refresh.

If notify == 1, the module will follow with a GET command. If notify == 0, the module may follow with a GET command (to validate that an attacker hasn't refreshed a session). Notify is OPTIONAL and defaults to 1. If the mobile client hasn't registered with the module, the module will follow with a key exchange.

400: Bad Request (Json Parse failure in request)

403: Forbidden - lan_ip on a different network

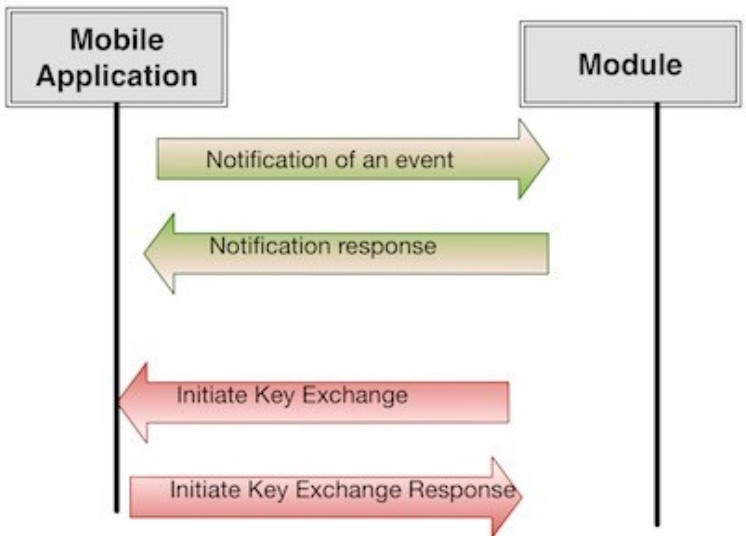
404: Not Found - Lan Mode is not enabled for this module

412: Precondition Failed - Module does not have lanip key

3.2.5 Key Exchange

Successful completion of the local registration process/key exchange means the Module and Mobile Application will exchange property value changes over the local WLAN. See Local Registration for more details.

Figure 2: Key Exchange and Notification Activities



The Module initiates a key exchange with the Mobile Application.

```
$ curl -X POST -d '{"key_exchange":{"ver":1,"random_1": "...","time_1":
1234567,"proto": 1,"key_id": 1}}'

 -H "Content-Type: application/json"
http://192.168.0.2/local_lan/key_exchange.json
```

Where the Module sends:

- ver - The version of this security exchange

- random_1 - A base-64 encoded random byte string, up to 16 characters long after encoding.
- time_1 - Module current time-related positive integer of up to 15 digits.
- key_id - sequence number identifying the shared secret.
- proto - proto number denotes encryption/hmac ciphers and functions
  - 1 = "CBC_AES256_SHA256"
  - other values are reserved for future use

Response:

```
{
  "random_2" :  "...",
  "time_2" :  456788999
}
```

The mobile app responds with:

- random_2 - A 16 byte string of random alpha-numeric characters
- time_2 - Mobile Application current time-related positive integer of up to 16 digits.

For any non-200 status code, the module deletes the client from the list of lan clients and reissues a session request for the key exchange to begin

Application Status Codes:

200: OK - The key exchange was successful and local registration is successful.

400: Bad Request - The json object could not be parsed.

403: Forbidden - A commands request was received without an established session.

404: Property Not Found: The property does not exist in the application template

405: Method Not Found - Command request must be a GET request

412: Precondition Failed - The key_id did not match.

424: Method Failure - General key exchange failure.

426: Upgrade Required - The prototype number for cyphers/hashes is not supported.

495: Cert Error - The client could not generate keys.

503: Service Unavailable - Insufficient resources or maximum number of sessions exceeded.

3.2.6 Session Keys

With the key exchange complete, the session keys are derived. Both the Module and the Mobile App use the concatenation of random data + times + a constant as the 'known data' to generate the HMAC. The method, as described, uses these elements in different order to create the keys for both parties. A varying constant is included to improve security for each session based on security element.

Algorithm is described in RFC 5246, Section 5 (TLS PRF with SHA256 as HMAC)

key = PRF(shared_secret, seed_value)

where PRF(secret, seed) = HMAC_hash(secret, A(1) + seed)

where A(0) = seed, and A(i) = HMAC_hash(secret, A(i-1))

where HMAC_hash is SHA256

where "+" denotes concatenation

so:

key = HMAC_SHA256(secret, HMAC_SHA256(secret, seed) + seed)

Seed value is different depending on which key is being generated:

- Mobile Application uses the data passed from the Module

  Signing key: <random_1> + <random_2> + <time_1> + <time_2> + 0

  Encrypting key: <random_1> + <random_2> + <time_1> + <time_2> + 1

  IV CBC seed: <random_1> + <random_2> + <time_1> + <time_2> + 2

- Module uses the data passed from the Mobile Application

  Signing key: <random_2> + <random_1> + <time_2> + <time_1> + 0

  Encrypting key: <random_2> + <random_1> + <time_2> + <time_1> + 1

  IV CBC seed: <random_2> + <random_1> + <time_2> + <time_1> + 2

NOTE	The final 0, 1, 2, seed values are the ASCII values 0x30, 0x31, 0x32 The IV seed is the upper/left-most 16 bytes of the KDF result.

All data exchanges on the secure session are structured in a JSON object of the following format:

{ "enc": "...", "sign": "..." }

- enc - encrypted (sequence number + clear text JSON object)
  - {"seq_no": "...", "data":<...>"}

    seq_no - sequence number of the message

    data - the clear text deserialized JSON object
- sign – signature

The UTF-8-encoded JSON string is encrypted using CBC with AES 256 (or the cipher defined by proto in the key exchange) where IV was calculated at key exchange time using the KDF.

The NUL termination of the JSON string is considered part of the plain text to be encrypted.

The encrypted {seq_no & data} are then base64 encoded. The result is value associated with "enc" in the JSON object format above.

The signature is created by taking the clear text data value above (without the NUL termination) and then applying HMAC to it as described in RFC 2104, using SHA256, (or the HMAC defined by proto in the key exchange), as the hashing function. The result of the HMAC function is base64 encoded and is the value associated with "sign" in the JSON object format above.

Verification of the message on the other end is done in reverse. The flow is depicted below:

Encode

enc: {seq_no & data} - > UTF-8 > encrypt > base64.encode

sign: enc.signature > base64.encode

Decode

enc: base64.decode > decrypt > UTF8 > {seq_no & data}

sign: base64.decode > enc.signature

An incrementing seq_no is required to harden the signature by ensuring that is based on non-static data. The seq_no is informational at this time: it is not used for any other security or networking purpose.

Cipher specifics:

1.0 Standard AES 256 CBC no padding

1.1 The crypto key is 256 bits

1.2 The IV is 16 bytes to match the 128 bit AES block size

1.3 Cipher block chaining (CBC) based on:

encrypt first time:

iv XOR clearText -> AES -> cipherText

encrypt subsequent times:

prior cipherText XOR clearText -> AES -> cipherText

decrypt first time:

cipherText -> AES -> XOR iv -> clearText

decrypt subsequent times:

cipherText -> AES -> XOR prior cipherText -> clearText

1.4 The chaining cycle begins when the keys and seeds are generated.

1.5 Blocks less than 16 bytes are padded out with 0x00.

2.0 The signature is calculated on the clearText of {seq_no & data}.

2.1 The signature calculation does not include the nul termination.

2.2 The clearText must contain a nul terminator prior to encryption.

3.2.7 Payload Buffer Sizes

Here is a breakdown of the max buffer length required for encryption/decryption at various stages.

Format, Formulas, and Assumptions Used:

AES256 Encryption

Payload Format: {"enc":"x","sign":"y"}

Encryption Format:
{"seq_no":x,"data":{"properties":[{"property":{"base_type":"string","value":"y","name":"z"}}]}

cipherLen = (round_up(clearLen/16) + 1) * 16

base64Len = 4 * ((inlen + 2) / 3) + 1

Sequence # is u16 so max length is 5

Value has a max length of 256

Name has a max length of 27

Calculations:

Max length of Encryption:

Max Cleartext Length = 92 + 5 + 256 + 27 = 380

Max Encryption Length = 400

Base 64 Length Needed = 537

Max Length of Sign:

Sign is ALWAYS 32 bytes

Base64 Length = 49 bytes

Total Payload Size: 20 + 49 + 537 = 606

3.3 WLAN Local APIs

It's important to recognize the JSON objects in this section are encapsulated as secure data before they are exchanged over the WLAN. Refer to the Payload Encapsulation section for details.

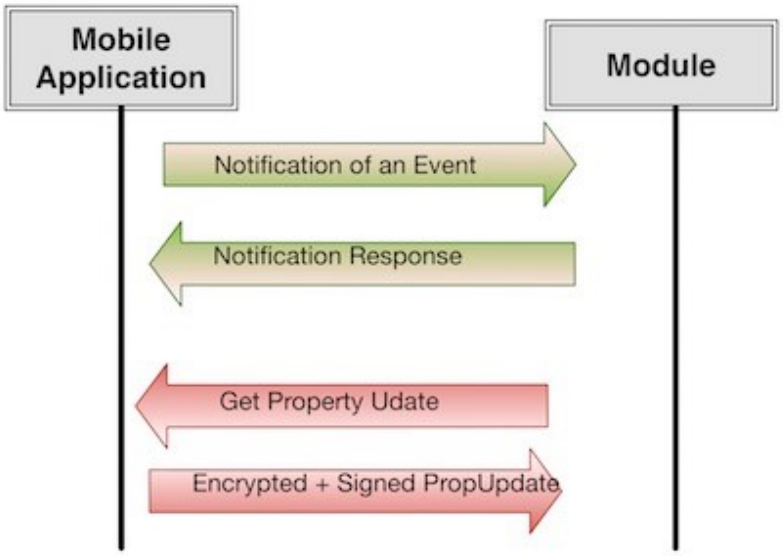
3.3.1 Command Transactions

The Mobile Application initiates secure communication session with a Module using the POST local_reg API as documented in the Notification section. A secure session is defined as the time beginning at successful key exchange, and ending after the defined keep-alive period or receiving a DELETE request as described below.

The Module continues to send property changes to the Device Service. The Mobile Application only sends property changes to the Module while the session is active.

Sometime after the Mobile Application sends a Notification, the Module requests commands and properties from the Mobile Application. If there are any commands in the GET response, the response to those commands is sent back to the Mobile Application using a PUT.

Figure 3: Property and Event Activities



The GET commands request returns a JSON object containing objects "cmds" and/or "properties", either of which can be empty or absent. Note that the encapsulation isn't shown here, but the response is encapsulated as shown in Payload Encapsulation above.

```
$ curl http://192.168.0.2:10275/local_lan/commands.json
```

Response:

```
{
  "cmds" : [
   {
    "cmd": {
    "id": "<cmd-id>", (unsigned 32-bit int)
    "method": "<cmd-method>",
    "resource": "<cmd-resource>",
    "data": "<data-for-cmd>",
    "uri": "<uri-for-response>"
    }
  },
   ...
 ]
}
OR
{
  "properties": [
   {
    "property": {
```

```
      "name": "<property-name>",
      "value": "<value>",
      "base_type": "<type>"
     }
    },
...
 ]
}
```

The Mobile App may respond to this GET with one of the status codes below:

Status Codes:

200: OK

204: No Content - The properties and/or cmds arrays are empty

206: Partial Content

There are more properties and/or cmds enqueued. The Module needs to issue a GET commands request when processing of the current one is completed.

401: Unauthorized - Decryption failed (Module drops the session)

404: Not found (module doesn't drop the session, the app must send another notify if it wants the module to do a GET)

xxx: Any other status code causes module to drop the session

This response may contain "cmds" or "properties", but not both, and for now, only one command per payload is permitted.

If any commands require a response, the module sends the encapsulated response with a POST to the specified URI.

```
curl -X POST -d '{"name":"Blue_button","value":1}' \
-H "Content-Type: application/json" \
http://192.168.0.2:10275/<uri_given_in_cmd>[?cmd_id=%d[&status=<status_id>]
```

Refer to the section Get All Property for details on the above URL POST.

Response:

In response to this POST, the Mobile App should return "200 OK". However, the status returned is ignored by the module, since there is no recovery possible if the module has been given the improper URI or the Mobile App is unable to parse the result.

3.4 Delete Session

To delete the session, the Mobile App starts the command transaction in the usual fashion, and provides the command "DELETE".

```
$ curl http://192.168.0.2:10275/local_lan/commands.json
```

Response:

```
{
  "cmds" :  [
   {
    "cmd": {
   "cmd_id": <any integer>, (will be ignored)
      "method": "DELETE",
      "resource": "local_reg.json",
      "data": <anything>, (will be ignored)
      "uri": "local_lan" (will be ignored)
      }
   }
    ]
}
```

The module deletes the session. No response back to the mobile app is necessary.

3.4.1 FIN Message

The module may end an ongoing LAN session for a multitude of reasons:

6. Decryption/signature failure for a packet from the app
7. App didn't refresh its session in time
8. Bad status code received when the module did a POST/GET operation on the app

To inform the app that the module is ending the session, it sends a POST message to the app in the following manner:

```
$ curl -X POST -d '{"seq_no":3,"data":{"dsn": "AC000W000001739"}}' \
-H "Content-Type: application/json" \
http://192.168.0.2:10275/local_lan/fin.json
```

The app can now choose if it wants to re-establish the LAN session with this module. Note that this FIN message is not done if the app deletes the session using the "DELETE" command, see above Delete Session.

3.4.2 Get All Properties

After session establishment, the Mobile App requests the current values of all properties from the Module. To retrieve all properties and associated current values the Mobile App issues the following command:

```
$ curl http://192.168.0.2:10275/local_lan/commands.json
```

Response:

```
{
   "cmds" :  [
   {
```

```
      "cmd": {
       "cmd_id": <unique_cmd_identifier>,
       "method": "GET",
       "resource": "properties_lan.json",
       "data": <anything>, (will be ignored)
       "uri": "local_lan/property/datapoint.json"
      }
    }
   ]
}
```

It is possible that the module may have some regular property updates in the middle of the command responses. The App should be able to handle this and only allow the user to start toggling property values after it receives an EOF on the command response. While the module is in the middle of posting responses to the "get all properties" command, it does not perform a GET on the app.

3.4.3 Get Property

Depending on the app layout, it might be more efficient for the app to request the values of individual properties instead of all properties. To do this, the Mobile issues the following command:

```
$ curl http://192.168.0.2:10275/local_lan/commands.json
Response:
{
  "cmds" : [
   {
    "cmd": {
    "cmd_id": <unique_cmd_identifier>,
    "method": "GET",
    "resource": "property.json?name=Blue_LED",
    "data": <anything>, (will be ignored)
    "uri": "local_lan/property/datapoint.json"
   }
  }
 ]
}
```

Responses:

200: OK

204: No Content - The properties and/or cmds arrays are empty

206: Partial Content

There are more property requests enquired. The Module needs to issue a GET command request when processing of the current one is completed.

401: Unauthorized - Decryption failed

3.4.4 Property Changes

Mobile Application

When the Mobile Application wants to send properties to the module, it initiates the command sequence in the usual way, as described in Command Transactions.

The response to the module's GET contains encapsulated property names and values as shown below.

```
$ curl http://192.168.0.2:10275/local_lan/commands.json
```

Response:

```
{
  "properties": [
   {
     "property": {
    "base_type": "boolean",
    "value": 1,
    "name": "Blue_LED"
    }
   },
   {
    "property": {
    "base_type": "boolean",
    "value": 0,
    "name": "Green_LED"
   }
  }
 ]
}
```

Status codes:

200: OK - normal response

401: Unauthorized - Authentication failed

429: Too Many Requests - The Mobile App's server is busy with a prior request.

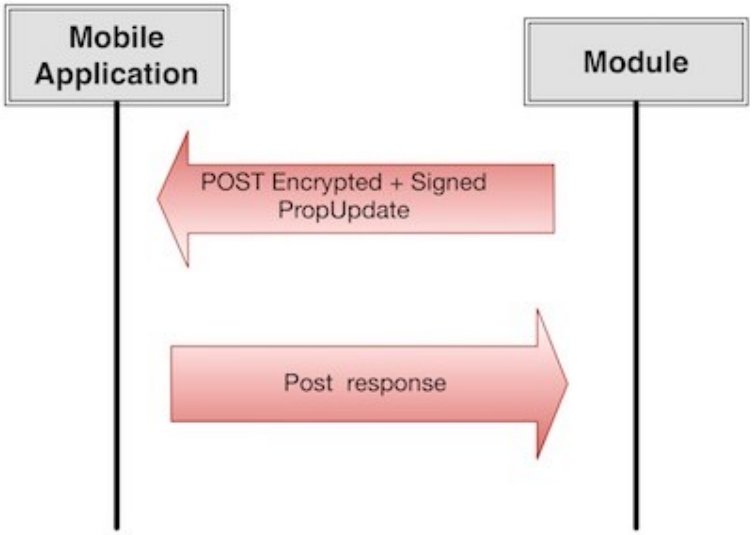
The application needs to reinitiate the GET request to the module.

If the module receives a non-200 return code it deletes the app registration. The app needs to re-register.

Module

When a property changes on the Module, it does a datapoint PUT to the Mobile Application server that encrypted and encapsulated.

Figure 4: Post Activities




```
$ curl -X POST -d '{"name":"Blue_button","value":1}' \
-H "Content-Type: application/json" \
http://192.168.0.2:10275/<uri_given_in_cmd>[?echo=true][?cmd_id=%d[&status=
<status_id>][&eof=true]]]
```

The *?echo=true* is a parameter used by the module to inform the mobile app this datapoint is an echo from the server. If the *?echo=true* parameter is omitted, the PUT is not an echo. By convention, a PUT on a "to device" property is an echo.

The *?cmd_id=%d* parameter is always included in response to a command requests. This aids in matching transaction requests and responses. For each command, an http status code is returned as documented below. When the final PUT of the transaction is sent, the *eof=true* parameter is included to flag, this is the final datapoint in the series. The eof=true parameter is only applicable for commands that require multiple PUTs as a response, that is the properties_lan.json command.

status_id from Module:

200: OK - command succeeded

404: Not Found - command not found or property name not found

503: Service Unavailable - MCU is taking too long to respond for the value of the property (timeout)

Status Codes from Mobile Application:

200: OK - normal response

400: Bad Request: the payload was not as expected/parseable

401: Unauthorized - Authentication failed

404: Property name not recognized (module doesn't drop the session, the app must send another notify if it wants the module to do a GET)

xxx: Any other status code causes module to drop the session

3.5 Keep-Alive/Power Management

The two APIs created to notify the Module that it is ready to receive property notifications and are also used to minimize power usage.

The Extend Session call should only be used when the mobile app is active and in the foreground.

The Delete Session call should only be used when the Mobile Application moves to an inactive state (in the background, exit, and so on). ,

The Mobile Application must extend the notification period, within the keep-alive timeout window to continue receiving property updates from the Module. When the Mobile Application moves to the background, it should kill the secure session call to maximize power savings on the Module.

4250 Burton Drive, Santa Clara, CA 95054
Phone: +1 408 830 9844
Fax: +1 408 716 2621