Cassia Software Development Kit (SDK) for S1000, S1100 and X1000

This document shows how you can use the Cassia SDK to integrate your Bluetooth end devices with the Cassia S1000, S1100 and X1000 Bluetooth router, without requiring any changes to the Bluetooth end devices. Step-by-step instructions, RESTful APIs, debug tools, and basic troubleshooting procedures are included.

Table of Contents

1 Overview

The Cassia Bluetooth Router is the world's first long-range Bluetooth router designed for enterprise deployments, enabling seamless coverage of any size and scale. It extends Bluetooth range up to 1,000 feet, open space, line-of-sight, and enables remote control of multiple Bluetooth Low Energy (BLE) devices without requiring any changes to the Bluetooth end devices themselves.

The Cassia Software Development kit (SDK) was developed to enable third party developers and device manufacturers to utilize the Bluetooth routing and extended range capabilities of the Cassia Router while using their Cloud services to connect and control multiple BLE devices per Router simultaneously. Furthermore, the Cassia SDK is designed to integrate directly into your application/server using an HTTP/HTTPS based communication protocol, which provides programming language flexibility. Cassia supports C#, Node.js, and Java, but you can choose other languages as you prefer. This document helps you learn how to use the Cassia SDK and its associated services.

The Cassia SDK for S1000, S1100 and X1000 is built into the Cassia IoT Access Controller (AC) providing the following functions:

a. Connect and control your BLE devices.
b. Support three modes: Scanning, Connecting, Broadcasting.
c. Write/read data to/from the BLE device via the Cloud server.
d. Read data as notification/indication events from the BLE device via the Cloud server.
e. Support multiple uplink networking interfaces to flexibly adapt to various environments.

2 Architecture Diagram

2.1 What is the Cassia Access Controller?

The Cassia IoT Access Controller (AC) is a powerful IoT network management solution. It provides RESTful APIs for the business to do data collection, positioning, roaming, and security policy management, enabling the remote control of Cassia Bluetooth routers across the Internet.

2.2 SDK Working Diagram

You can operate your BLE devices via the AC's access to the Cassia routers, using a set of RESTful APIs. The Cassia router is a Linux-based thin Access Point (AP). Please see Figure 1 for the Cassia SDK Working Diagram, using S1000 as an example.

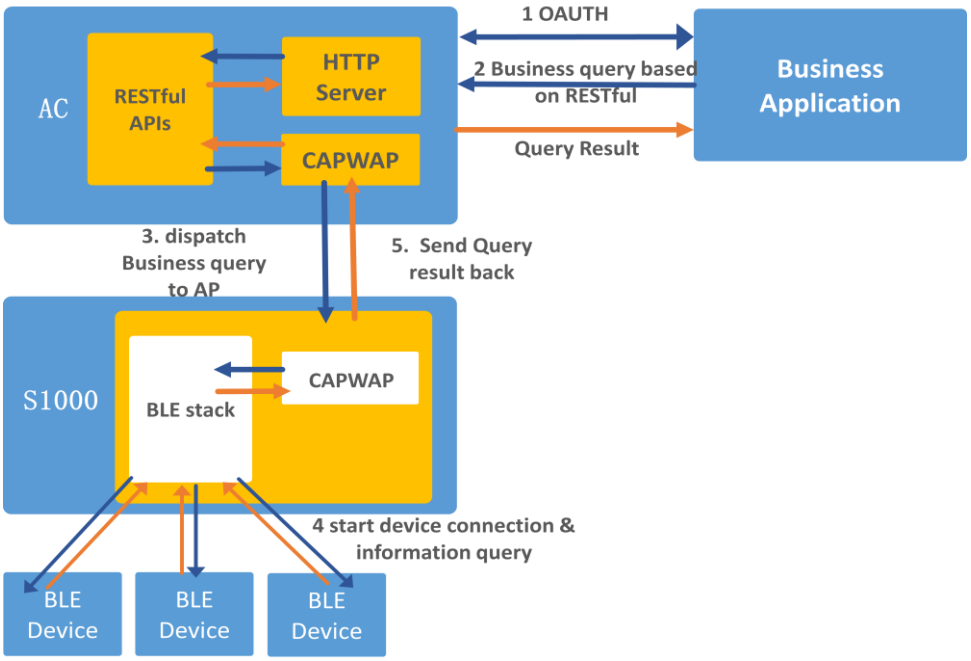

Figure 1: Cassia SDK for S1000 Working Diagram

The business application initiates an OAuth authentication request (generated using its developer ID and secret) to the Cassia Access Controller first. Once the authentication succeeds, it will send an HTTP query based on Restful to the AC. Next, the AC dispatches the query to a corresponding Bluetooth router via encrypted CAPWAP. The router then executes the query upon the BLE devices, and passes the result back to the AC, and then the business application.

3 Prerequisites

Once you have the Cassia Bluetooth Router and SDK, you will also need:

a) Developer account credentials

You will need a Developer Key and a Developer Secret. These developer credentials authorize the remote control of the Cassia Bluetooth router. To request your developer account credentials, contact the Cassia support team at support@cassianetworks.com or your sales representative.

Here is a sample credential:

```
client_id:tester, secret:198c776539c41234
```

b) License key

To use your Bluetooth Router with your AC, you will also need a license key. The license key governs the number of Bluetooth routers that can be managed by the AC, for how long and features available. Contact the Cassia support team at support@cassianetworks.com or your sales representative. Here is an example of a license key:

```
v009-8q61-jy6a-l9fe-2gz8
```

c) Open UDP ports 5246 and 5247 on your Wi-Fi router or Firewall

The ports that Cassia routers use to communicate with the Cassia Cloud are UDP 5246 and 5247. If you have a Firewall or Wi-Fi Router that blocks those ports, please make sure to open them.

4 Getting Started

4.1 Step One: Associate your Bluetooth Router with the AC

Plug in your Cassia router via Ethernet cable and power it on. Locate your IP address of the router from your network switch or DHCP server. Enter your IP address in a web browser and load web console. In Common section, please enter your Developer Key and AC Address as shown below in Figure 2.

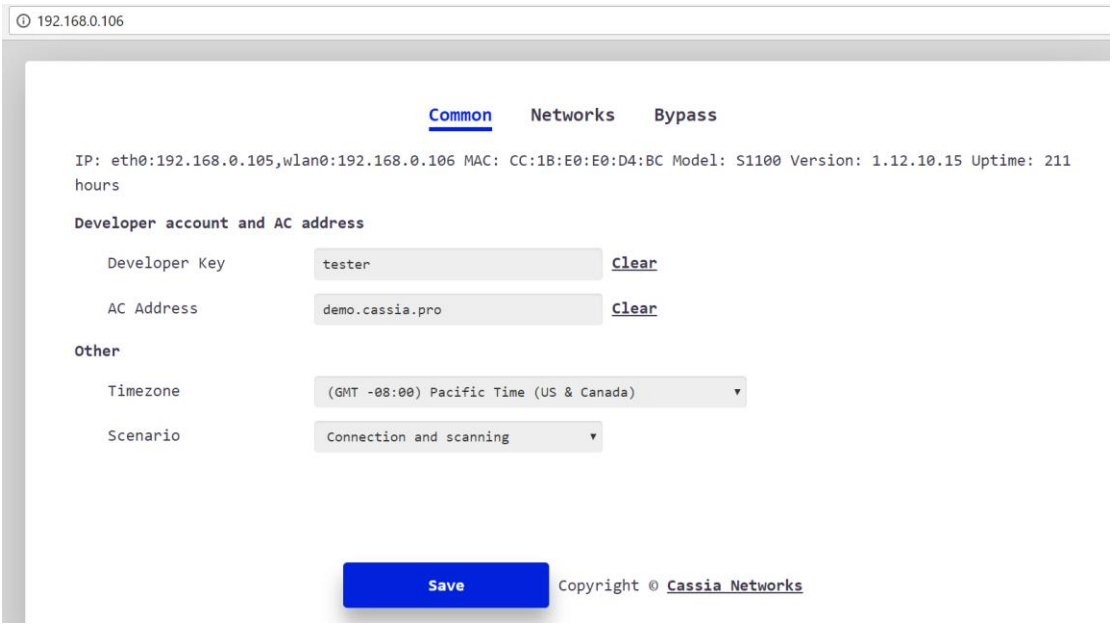

Figure 2: Input your developer key and AC Address

Note: You can skip this step if you are deploying your AC at the same subnet as your routers, or using DHCP option43 or DNS for AC auto discovery.

4.2 Step Two: Access Cassia IoT Access Controller (AC)

You have the option to deploy an AC either in the Cassia Cloud, your own private cloud or a hardware appliance. For set up, login to the AC and navigate to the Settings page. As noted in the screenshot below, input your Developer Key, Developer Secret, and License. (Figure 3).

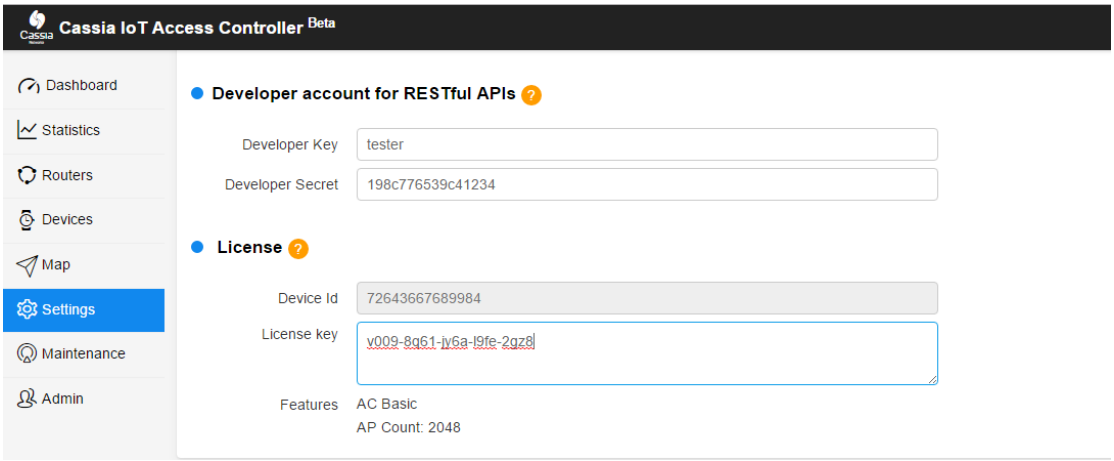

Figure 3: Input your developer key, secret, and license

4.3 Step Three: Configure your Bluetooth Router

Navigate to the Routers page, as noted below in Figure 4.

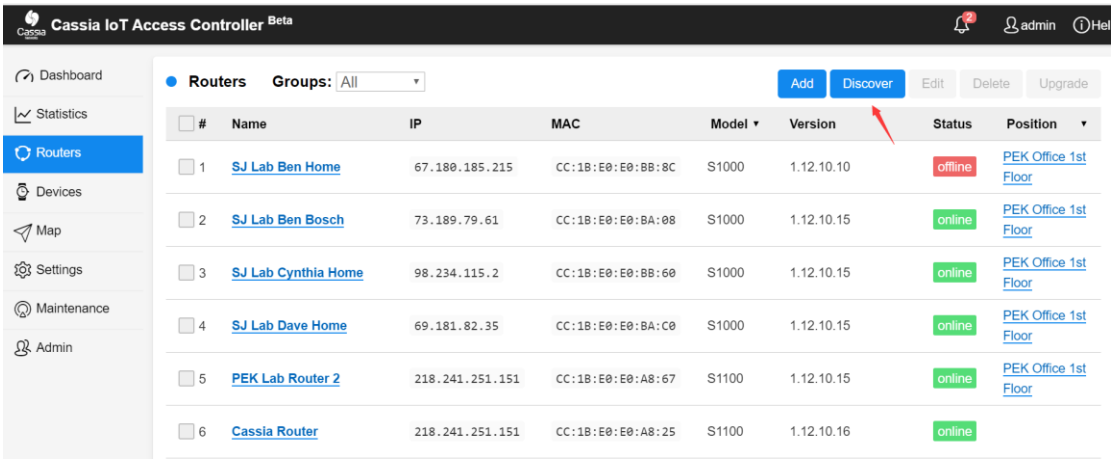

Figure 4: Cassia AC Routers Page

Search your router in the list using its MAC address. You can find your router's MAC address at the bottom of the router, as noted below in Figure 5.

Figure 5: Cassia S1000 router's MAC address

If your router is not listed in the Routers list, click the Discover button (near the top-right area of the interface), see Figure 6 to discover and add your router into the AC.

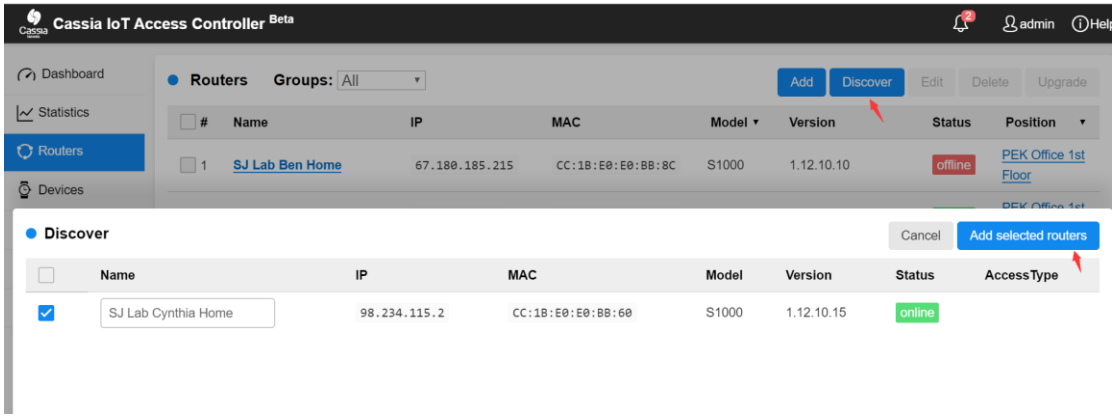

Figure 6: Discover your router and add it into the AC

4.3.1 Configure Common Settings

To configure your router, check the checkbox before it and click the Configure Button to get to the router's configuration page, as see in Figure 7. Make sure to enter the same Developer Key as your AC. If the AC Address field is empty, please enter your AC domain name or IP address, for example, demo.cassia.pro.

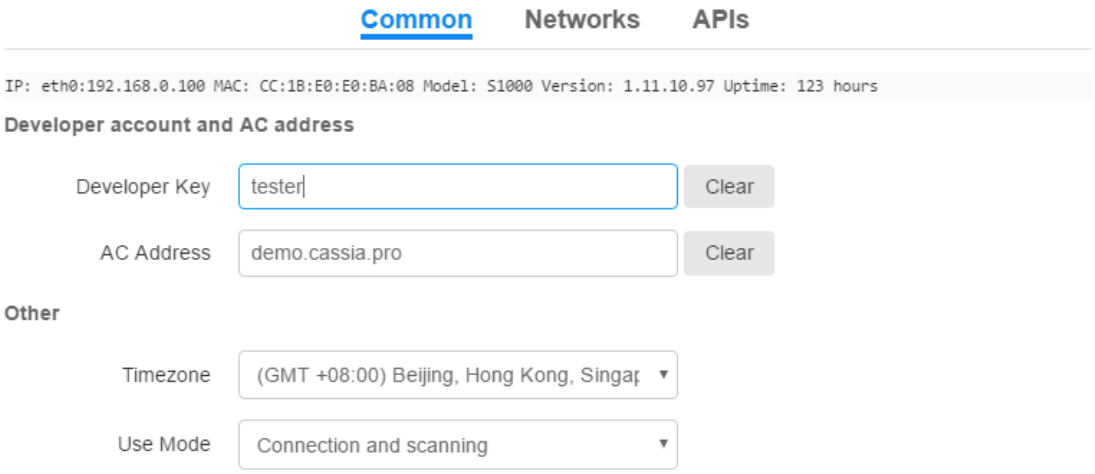

Figure 7: Set Developer Key on your Router

4.3.2 Configure Networks Settings

Next, click the Networks tab to setup your networking configurations. The Cassia Bluetooth Router supports the following networking uplinks.

- Wired: you can set it to use static IP or DHCP
- Wireless: enter your WiFi SSID and password, click Save. The router will go offline briefly and back online again.
- 3G/4G dongle: currently we support two cell dongles (ZTE MF823 in China, AT&T AirCard 313U in US)
- Networks Priority: When two or more network connections are activated, you can set priority levels for the networks. By default, the priority is: wired > wireless > 3G/4G.

4.3.3 Configure Bypass

You can configure your query data to bypass the AC and have it sent directly to a server you specify (aka, split horizon). To configure the bypass parameters, switch to the Bypass tab, see Figure 8 below.

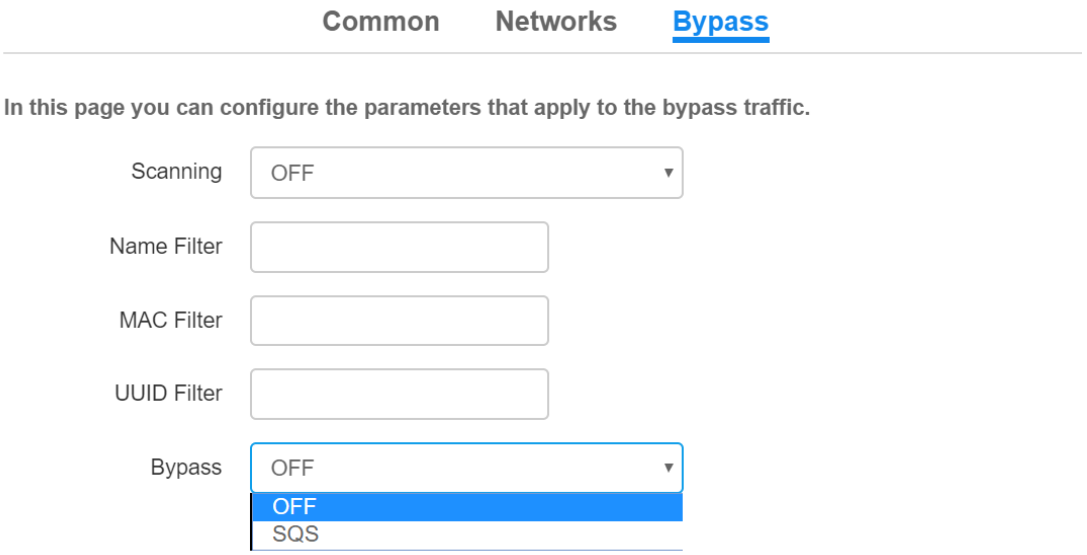

Figure 8: Set Bypass parameters on your Router

Currently, we only support SQS for bypass. MQTT is coming soon. Will update the document once it is available.

4.4 Step Four: Remote Control Your Router

Now you can operate your Router to do certain tasks with your end devices through RESTful APIs. If you would like to access your Router from the AC, you need to follow the steps below.

a. Do an OAuth2.0 authentication with the AC using client credentials granted. For example: you have a developer ID: tester, secret: 10b83f9a2e823c47, use base64 to encode string "tester:10b83f9a2e823c47" and get "dGVzdGVyOjEwYjgzZjlhMmU4MjNjNDc="

b. Authenticate the user identity using the following http request, taking demo.cassia.pro as your AC server as an example.

```
POST api/oauth2/token HTTP/1.1
    Host: demo.cassia.pro
    Headers: {Authorization: Basic dGVzdGVyOjEwYjgzZjlhMmU4MjNjNDc=
    Content-Type: application/x-www-form-urlencoded}

    Body:
    {grant_type=client_credentials}
```

If everything is correct, you will get a response like this:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{ token_type: 'bearer',
   access_token:
'2b6ced831413685ec33204abc2a9a476310a852f53a763b72c854fd7708499f1bc0b362
6bfcfef2a2cfe0519356c9d7cb1b514243cb29f60e76b92d4a64ea8bd',
   expires_in: 3600 }
```

c. Now you can use access_token to access the other RESTful APIs by appending an access_token parameter. For example, http://demo.cassia.pro/api/gap/nodes?event=1&mac=xxx&access_token=xxx. Or you can add {Authorization : 'Bearer ' + access_token } in headers.

5 RESTful APIs:

In the Cassia SDK, most of the Bluetooth GAP/GATT operations are exposed in RESTful APIs. The signatures of those APIs are fully compliant with Bluetooth SIG's Internet Working Group RESTful API specification. You can use the APIs on the Cassia Access Controller to interact with the Router, e.g. http://demo.cassia.pro/api/....

5.1 Interface Parameters

RESTful Interface parameters are defined below:

- mac: the mac address of a Cassia hub (eg： CC:1B:E0:E0:24:B4)
- node: the mac address of a BLE device (eg： EF:F3:CF:F0:8B:81)
- handle: after you find the device services, based on the device's Bluetooth profile, you can identify its corresponding handle index in the UUID (eg: 37)
- value: the value written into the handle
- chip: 0 or 1, indicates which chip of the Cassia hub is being used to operate this command. This parameter is optional. (Note: S1000 and S1100 only support chip 0, X1000 supports 0 and 1)
- interval: advertise interval, data type is number
- ad_data: advertise package, data type is string
- rsp_data: advertise response package, data type is string

5.2 Commonly Used APIs

a. How to use Cassia Router to scan for Bluetooth devices in its range?

To use the Router to scan for Bluetooth Low Energy (BLE) devices using your AC's endpoints:

```
GET http://{your AC domain}/api/gap/nodes?event=1&active=&mac=<hubmac>
```

This API is a SeverSide Event (SSE) which will be running continuously. Refer to section 6 for the details of SSE.

- active: 0 or 1, 0 indicates passive scanning and 1 active scanning. This parameter is optional. If you don't specify, by default S1000 will perform passive scanning.
- filter_duplicates: 0 or 1, turn on/off to filter duplicated records.

b. How to filter scanned data based on MAC address, device name or others?

This is a useful API which can significantly reduce the amount of traffic sent from the Router to the server.

```
GET http://{your AC domain}/api/gap/nodes?event=1&mac=<hubmac>&filter_mac=<mac>

http://{your AC domain}/api/gap/nodes?event=1&mac=<hubmac>&filter_mac=<mac1>,<mac2>,<mac3>

http://{your AC domain}/api/gap/nodes?event=1&mac=<hubmac>&filter_name=<name1>,<name2>
```

You can also filter out devices based on its RSSI level, e.g. Filter out devices RSSI value weaker than a certain value.

```
GET http://{your AC domain}/api /gap/nodes?event=1&mac=<hubmac>&filter_rssi=<num>
```

In addition, you can filter out a device based on its service UUID inside its advertise packet.

GET http://{your AC domain}/api /gap/nodes?event=1&mac=<hubmac>&filter_uuid=<uuid>

Note: above APIs related to filtering only apply for the traffic going through the Cassia AC. (APIs for bypass traffic filtering are coming. When added, we will update the document reflect that they are available.)

c. How to connect/disconnect to a target device?

To use the Router to connect to specific BLE devices using Cassia demo AC's endpoints:

- Timeout: in ms, the connection request will timeout if it can't be finished within this time. For S1000/S1100, the timeout is set to 20000ms and can't be configured. For X1000, this parameter is configurable.

POST http://{your AC domain}/api/gap/nodes/<node>/connection?mac=<hubmac>

To disconnect:

DELETE http://{your AC domain}/api/gap/nodes/<node>/connection?mac=<hubmac>

Get the device list connected to a router:

GET http://{your AC domain}/api/gap/nodes?connection_state=connected&mac=<hubmac>

d. How to discover GATT services and characteristics?

Discover all services:

GET http://{your AC domain}/api/gatt/nodes/<node>/services?mac=<hubmac>&all=1

Discover all characteristics

GET http://{your AC domain}/api/gatt/nodes/<node>/characteristics?mac=<hubmac>&all=1

Discover all characteristics in one service

GET http://{your AC domain}/api/gatt/nodes/<node>/services/<service_uuid>/characteristics

Discover all descriptors in one characteristic

GET http://{your AC domain}/api/gatt/nodes/<node>/characteristics/<characteristic_uuid>/descriptors

Discover a specific service by service UUID:

GET http://{your AC domain}/api/gatt/nodes/<node>/services?mac=<hubmac>&all=1&uuid=<uuid>

Discover a characteristics by characteristics UUID:

GET http://{your AC domain}/api/gatt/nodes/<node>/characteristics?mac=<hubmac>&uuid=<uuid>

Discover all services, characteristics, and descriptors all at once:

GET http://{your AC domain}/api/gatt/nodes/services/characteristics/descriptors?all=1

e. How to read/write the value of a specific characteristic?

The read/write operations are based on the handle of a specific characteristic.
The handle of a specific characteristic can be found in the discovery result.
To read by handle:

GET http://{your AC domain}/api/gatt/nodes/<node>/handle/<handle>/value?mac=<hubmac>

To write by handle:

GET http://{your AC domain}/api/gatt/nodes/<node>/handle/<handle>/value/<value>?mac=<hubmac>

f. How to subscribe to a notification/indication to a specific characteristic?

If you need to open a specific characteristic's notification or indication, you need to call the "discover service" interface first. To do so, find the corresponding descriptors of the specified characteristic, open the descriptors, find the UUID that contains "00002902" and its corresponding handle. Use this handle for the API call. To open the notification, set the value to "0100"; to open the indication, set the value to "0200"; to close the notification / indication, set the value to "0000".

GET http://{your AC domain}/api/gatt/nodes/<node>/handle/37/value/0100?mac=xxxx

g. How to Reboot a Router remotely?

GET http://{your AC domain}/api/cassia/reboot?mac=<hubmac>&access_token=XXX

h. How to get advertise data?

- interval: advertise interval
- ad_data: advertise package, data type is string
- rsp_data: advertise response package, data type is string

Begin advertise data:

GET http://{your AC domain}/api/advertise/start?mac=&chip=&interval=&ad_data=&resp_data=

Stop advertise data:

GET http://{your AC domain}/api/advertise/stop?mac=&chip=&interval=&ad_data=&resp_data=

i. How to configure your router to use SQS bypass mode (aka, send SQS message to a third party server)?

Use the following new API to input SQS parameters first.

GET http://{your AC domain}/api/bypass/config?type=sqs&keyId=&key=&queueUrl=

Then add bypass=1 in a scanning.

GET http://{your AC domain}/api/gap/nodes?event=1&bypass=1&mac=<hubmac>

To monitor the status of bypass traffic, you can use the following API to obtain the number of SQS messages sent from the router in the last 30 seconds. It will return the number of success, number of failure, number of cancelled, and number of scanned.

GET http://{your AC domain}/dashboard/sqs?mac=<hubmac>

j. How to obtain the location info of your device?

Cassia supports room-based Bluetooth location tracking. Below are the related APIs.

To identify the closest router a BLE device is located:

```
GET http://{your AC domain}/api/middleware/position/by-device/<device_mac>
```

It will return {“hubMac”:”hubMac1”}, e.g. {“hubMac”:”CC:1B:E0:E0:01:47”},

To obtain the closest router list for all the BLE devices that the AC can detect:

```
GET http://{your AC domain}/api/middleware/position/by-device/*
```

It will return a list:

```
 {
      “device1”:{“hubMac”:”hubMac1”},
      “device2”:{“hubMac”:”hubMac2”},
      …
   }
e.g.：  {
            “11:22:33:44:55:66”:{“hubMac”:”CC:1B:E0:E0:01:47”},
            “11:22:33:44:55:77”:{“hubMac”:”CC:1B:E0:E0:01:48”},
}
```

To get the list of BLE devices around a Cassia router:

```
GET http://{your AC domain}/api/middleware/position/by-ap/<hub_mac>
```

It will return [“device1”,”device2”,”device3”…], e.g. [“11:22:33:44:55:66”,”11:22:33:44:55:AA”…].

To get the list of BLE devices for all the routers within the AC:

```
GET http://{your AC domain}/api/middleware/position/by-ap/*
```

```
It will return：
{
            “hubMac1”:[ “device1”,”device2”,”device3”…],
            “hubMac2”:[“device1”,”device2”,”device3”…],
            ….
}
e.g.：{
            “CC:1B:E0:E0:11:22”:[“11:22:33:44:55:66”,”11:22:33:44:55:AA”…],
            …
}
```

Note: For detailed API syntax, please refer to http://docs.cassiasdk.apiary.io/. We have sample code in iOS/Java/nodejs/js/C# for your reference. Please ignore the rest in that online document which was written for another product.

6 Server Sent Events (SSE)

6.1 What is SSE?

SSE is a technology where a browser receives automatic updates from a server via an HTTP connection. The Server-Sent Events EventSource API is standardized as a part of HTML5[1] by the W3C. SSE is used to send message updates or continuous data streams to a browser client. It needs to be manually terminated, otherwise it will keep on running until an error occurs.

Each SSE response starts with "data:". When debugging, you can input the URL of a SSE into a web browser. In the program, an SSE request won't return any data if you call the interface like a normal HTTP request, because a normal HTTP request only returns output when it finishes. In addition, when calling an SSE, you should monitor this thread, in case it is interrupted by an error or any unexpected incident, then you can restart it.

6.2 SSE in RESTful APIs

Three of the RESTful APIs are using SSE: scan, get device connection status, receive indication and notification.

a. Scan: See the following scan call as an example:

http://{your AC domain}/api/gap/nodes?event=1&mac=CC:1B:E0:E0:24:B4&access_token=xxx

This API call will constantly return the scan data from a Router with MAC address CC:1B:E0:E0:24:B4, until you stop the thread.

Note: If you use tools like CURL for HTTP request, the tool will return data when the HTTP request ends. However, our scan API is an SSE which NEVER ends and sends data in a stream, so it will hang the page. You should add the following snippet:

```
if ($stream = fopen($url_for_scan, 'r')) {
    while(($line = fgets($stream)) !== false) {
        echo $line;
    }
}
```

b. Get device connection status

SSE to get the connection status of all the devices that have connected to a router:

```
http://{your AC domain}/api/management/nodes/connection-state?mac=<hubmac>
```

When a router status is changed from online to offline, or from offline to online, you will get a response.

c. SSE to receive notification and indication

```
http://{your AC domain}/api/gatt/nodes?mac=<hubmac>
```

6.3 Best Practice

For a best practice in implementing your application, use the following guidelines:

a. Setup an SSE http://{your AC domain}/api/cassia/hubStatus

(without any parameters, need to OAuth before being used)

- When a Router is online, you will get this response: {"status":"online","mac":" <hubMac>"}.
- When a Router is offline, you will get {"status":"offline","mac":"<hubMac>"}.

b. When a Router's status is changed to "online", check the Router's SSE, if the SSE exists, destroy it first, then setup a new SSE of notification like previous:

http://{your AC domain}/api/gatt/nodes?mac=<hubmac>

If a SSE does not exist, setup a new SSE.

c. When a Router's status is changed to "offline", you can destroy the previous SSE.

7 SDK Debug tool

This is a development tool for developers to integrate Bluetooth devices with the Cassia Bluetooth router. With this tool, we put the SDK APIs into a visual interface. After an API call, the tool will show you the response messages, allowing developers to quickly implement the integration using a programming language they are familiar with. This tool is available at http://www.bluetooth.tech/nativeHubControl/index.html

See Figure 9 below.

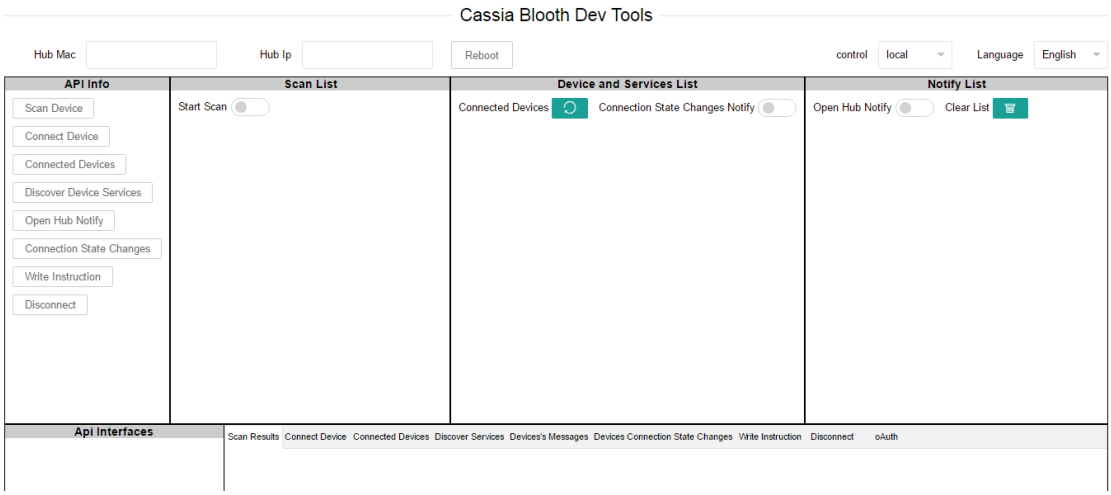

Figure 9: Cassia SDK Debug Tool

API Info: This area contains the user's most commonly used commands. Clicking on each button which will pop up parameters and descriptions related to that command.

Scan List: Turn on this option and the Cassia Router will start scanning for all BLE devices within its range. The BLE devices need to be in broadcast mode. You can connect to one or multiple devices from there. See below Figure 9 for an example.

Device and Services List: Turn on this option to see the connected devices. Based on the device's Bluetooth profile, you can write value or turn on the notifications. See below Figure 10 for an example.

Notify List: If you have turned on the notification of the correct handle, you will see a stream of raw data flowing in the Notify List window. See below Figure 10 for an example.

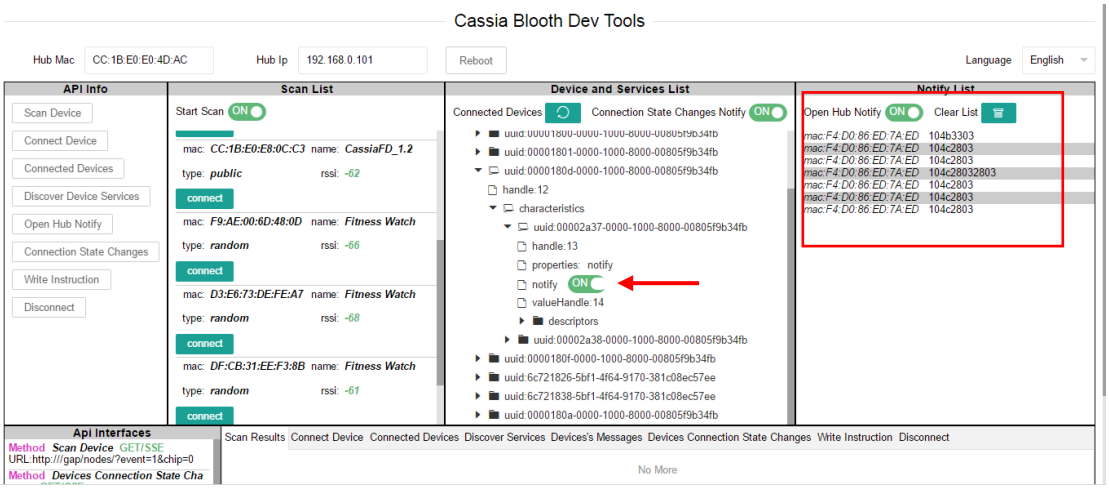

Figure 10 Cassia SDK Debug Tool example

8 Troubleshooting Error Messages

For HTTP 500 error, the following are the common errors.

1) "parameter invalid": wrong parameter value, such as, chip ID, MAC address or advertise type is wrong.
2) "device not found": it is possible that this device is disconnected. A GATT call to query the attribute of a disconnected device will return this error.
3) "memory alloc error": when the Bluetooth chip does not have enough memory to complete the operation it will return this error.
4) "operation timeout": each operation has a time-out value, especially those time-consuming operations, such as connection. When connecting to a device that does not exist, the operation will timeout after 20s.
5) "chip is not ready": this error will be reported when sending commands to the chip fails.
6) "chip is busy": when sending the same command to the chip, the system will return "chip is busy", such as at the beginning of a scan. All the GATT commands are mutually exclusive, continuous call will return this busy error.
7) "incorrect mode": Our S1000 only supports one role, either master or slave (due to the memory limit). These two roles are different, mainly reflected in the broadcast and scanning. When the router is a slave, it cannot conduct scanning; when the router is the master, it cannot send advertise which is connectable. If you set the unsupported parameters to the chip, the system will return this error.
8) "device not connect": same as "device not found" error.

9) "operation not supported": reserved for future use.
10) "failure": an error for all other failures not specified yet.
11) "need pair operation": some devices require an operation for pairing after a successful connection. If a GATT function call happens prior to the pairing, the system will Return this error.
12) "no resources": our Bluetooth chip can store the pair information up to 10s. If you pair too many devices, the system will report this error.
13) "Service Not Found": couldn't find a Characteristic inside a Service.
14) "type not supported": When set up Bypass scan, the protocol type you specified is not supported by this firmware.
15) "please set bypass params first": You have enabled to use Bypass mode, but haven't set the bypass parameters.

Appendix -- Sample code

```
var credentials = {
   id: 'tester',
   secret: '816213f8b5c2877d'
};

var access_token = '';

var request = require('request');
var options = {
      url : 'http://demo.cassia.pro/api/oauth2/token',
      method : 'POST',
      form : {'grant_type' : 'client_credentials'},
      headers : {
            Authorization : 'Basic ' + new Buffer(credentials.id + ':' + credentials.secret, 'ascii').toString('base64'),
      }
};

request(options, function(error, req, body) {
      if (error) {
            console.log(error);
            return;
      }

      var data = JSON.parse(body);
      access_token = data.access_token;
      console.log(data);

      var options = {
            url : 'http://demo.cassia.pro/api/client',    //you can change this to the IP address and port your Router using.
            method : 'GET',
            // form : {'grant_type' : 'client_credentials'},
            headers : {
                  Authorization : 'Bearer ' + access_token,
            }
      };
      });
});
request(options, function(error, request, body) {
      console.log(body);
});
});
```