BLE Provisioning

# Introduction

This application notes describes the use of multiple APIs to create a provisioning application using BLE as the mode of transferring provisioning data. The accompanying code sample helps understand BLE provisioning in detail.

# BLE Provisioning

The sample application in this document creates a BLE GATT profile and service which is then used to provision a connection to a Wi-Fi network. An android application running a custom android application is used to input the Wi-Fi credentials.

# Topology

Talaria TWO

BLE Mobile Application

Access Point

BLE

Wi-Fi

Provisioning

Figure : Topology

# List of APIs

1. bt\_gap\_init():Creates and initializes all the resources needed to run GAP service and must be called before using any of the other functions in the Bluetooth GAP interface.
2. common\_server\_create():Creates a server with the name, manufacturer name and appearance passed, and creates and adds below services to the created server:
3. Generic Access
4. Generic Attribute
5. Device Information Services

Moreover, common server instance is reference counted. Future calls to this API increment the reference count and returns the instance created in first call.

|  |
| --- |
| void common\_server\_create(char \*name, uint16\_t appearance, char \*manufacturer\_name); |

1. common\_server\_destroy():Decrements the reference count for common server created by API common\_server\_create(). If reference count reaches zero, it destroys the server created and frees up all the resources.

|  |
| --- |
| void common\_server\_destroy(); |

1. bt\_gatt\_create\_service\_128(): Creates a service declaration from a 128-bits UUID given as parameter and returns the pointer to the GATT service.
2. bt\_gatt\_add\_char\_16(): Adds a characteristic with a 16-bit UUID to a created service.
3. bt\_gatt\_add\_service(): Adds a created service to the local server list.
4. bt\_gap\_cfg\_adv\_set(): Configures the advertisement parameters for the GAP peripheral through which the frequency of advertisement transmission in fast and slow mode can be adjusted. It also configures the Tx power for advertisement and the channel map used.
5. bt\_gap\_addr\_set(): Sets address for given address type.
6. bt\_gap\_connectable\_mode(): Sets the device in desired connectable mode.
7. bt\_gap\_server\_link\_add(): Used to add a GATT server to the GAP connection.
8. bt\_gap\_server\_link\_remove(): Used to remove GATT server from the GAP connection.
9. wcm\_create():Create a Wi-Fi network interface.
10. wcm\_add\_network\_profile(): Adds a network profile to WCM.
11. wcm\_delete\_network\_profile(): Deletes the currently active network profile that was previously added.
12. wcm\_auto\_connect(): Starts or stops auto connect.
13. wcm\_notify\_enable(): Enables callbacks of link and IP address changes.

# Code Walkthrough

## Overview

The sample code in the path apps\ble\_provisioning\src\main.c implements a server called Inno\_BLEWiFiProvisioning. It creates and starts a custom GATT service with two write-only characteristics. These characteristics can be written by a connected BLE Client. This allows the BLE Client to send SSID and Passphrase to Talaria TWO. Whenever these characteristics are written by the client, relevant callback is received.

In addition to the custom service, the server also makes use of common server functionality provided by the BLE API. Specifically, this adds the Generic Access, Generic Attribute, and Device Information services to the server.

## Flow Chart

Figure : Flow diagram

GAP initialization

Add custom and common GATT services/characteristics

Start GATT server (Enables advertisement)

No

Yes

BLE connection

Start Wi-Fi connection process with received provisioning data

(SSID, passphrase)

Stop BLE advertising

Yes

Yes

No

No

Receive provisioning data

(SSID, passphrase)

Wi-Fi connection success

## Sample Code Walkthrough

### GAP Initialization

The server starts by initializing the GAP Service:

|  |
| --- |
| bt\_gap\_init(); |

The GAP API must be initialized before other functions in the GAP interface are called.

### Adding Common GATT Server Functionality

The server uses the this common\_server\_create() API which creates a server name Inno\_BLEWiFiProvisioning with a manufacturer name as Innophase Inc.

|  |
| --- |
| common\_server\_create("Inno\_BLEWiFiProvisioning", 0, "Innophase Inc"); |

### Adding Custom GATT Service & Characteristic

The server’s custom service is created, and characteristics are added to it. The macro function defines the UUIDs of custom characteristics.

|  |
| --- |
| #define UUID\_WIFI\_SSID\_16 0xAB34  #define UUID\_WIFI\_PASSCODE\_16 0xCD34  #define UUID\_WIFI\_STATUS\_16 0xEF34 |

custom\_server\_create() creates the "custom server" i.e., adds the custom service and characteristic.

|  |
| --- |
| srv.cust\_service = bt\_gatt\_create\_service\_128(UUID\_CUSTOM\_SERVICE);  bt\_gatt\_add\_char\_16(srv.cust\_service, UUID\_WIFI\_SSID\_16, ssid\_provision\_cb, GATT\_PERM\_WRITE, GATT\_CHAR\_PROP\_W); bt\_gatt\_add\_char\_16(srv.cust\_service, UUID\_WIFI\_PASSCODE\_16,  pass\_provision\_cb, GATT\_PERM\_WRITE, GATT\_CHAR\_PROP\_W);  bt\_gatt\_add\_char\_16(srv.cust\_service, UUID\_WIFI\_STATUS\_16,  status\_provision\_cb, GATT\_PERM\_READ, GATT\_CHAR\_PROP\_R);  bt\_gatt\_add\_service(srv.cust\_service); |

bt\_gatt\_create\_service\_128() function creates a GATT service with a 128-bit UUID.

bt\_gatt\_add\_char\_16 is used to add a characteristic with a 16-bit UUID to a service.

Callback function is provided as a parameter to this function, which will be called when the characteristic is accessed. Properties and permissions for the characteristic are also specified with this API.

Here, three such characteristics are added.

1. UUID\_WIFI\_SSID\_16 and the callback associated when accessing this characteristic - ssid\_provision\_cb().
2. UUID\_WIFI\_PASSCODE\_16 and the callback associated when accessing this characteristic - pass\_provision\_cb().
3. UUID\_WIFI\_STATUS\_16 callback associated when accessing this characteristic - status\_provision\_cb ()

SSID and PASSCODE, both have WRITE permission and property. STATUS has read permission and property.

Finally, bt\_gatt\_add\_service adds the service to the server.

### Starting BLE GATT Server

Once the server’s services and characteristics are set up, it starts using the start\_server function. gap\_ops\_t gap\_ops are the GAP option objects to be passed to GAP functions.

|  |
| --- |
| static const gap\_ops\_t gap\_ops = {  .connected\_cb = connected\_cb,  .disconnected\_cb = disconnected\_cb,  .discovery\_cb = NULL,}; |

start\_server() function initiates Bluetooth provisioning.

|  |
| --- |
| bt\_gap\_cfg\_adv\_t bt\_adv\_handle;  bt\_adv\_handle.fast\_period = 10240;  bt\_adv\_handle.slow\_period = 0;  bt\_adv\_handle.fast\_interval = 160;  bt\_adv\_handle.slow\_interval = 480;  bt\_adv\_handle.tx\_power = 0;  bt\_adv\_handle.channel\_map = BT\_HCI\_ADV\_CHANNEL\_ALL;  bt\_gap\_cfg\_adv\_set(&bt\_adv\_handle); |

bt\_gap\_addr\_set() sets the BLE address.

|  |
| --- |
| bt\_gap\_addr\_set(bt\_hci\_addr\_type\_random, SERVER\_ADDR); |

bt\_gap\_connectable\_mode() makes the server connectable and enables the advertisement.

|  |
| --- |
| bt\_gap\_connectable\_mode(GAP\_CONNECTABLE\_MODE\_UNDIRECT,  bt\_hci\_addr\_type\_random, addr\_type\_zero, address\_zero, &gap\_ops); |

To allow other devices to connect to the device via Bluetooth, there is a need to start advertising and making the device connectable.

Here, bt\_gap\_cfg\_adv\_set sets parameters for advertisement.

The parameters passed for configuring the advertisement are explained as follows:

1. adv\_fast\_period is set to 10,240ms which is nearest multiple of 10 seconds in 625µs units.

This implies, the fast advertising will be attempted for nearly 10 seconds (10.24s) when the advertisement is enabled, post which the slow advertisement will be attempted.

1. adv\_slow\_period is set to 0. This implies, slow advertisement will be attempted indefinitely and there is no time bound programmed after which advertisement should stop automatically.
2. adv\_fast\_int is set to 160, which entails (160\*625µs) = 100,000µs = every 100ms is the interval at which fast advertisement will be attempted.
3. adv\_slow\_int is set to 1,600, which entails (1,600\*625µs) = 1,000,000µs = once in every second will be the interval of slow advertising.
4. bt\_gap\_set\_adv\_data sets the advertisement data. It is for legacy advertisement.
5. bt\_gap\_addr\_set sets the BLE address and address type. The sample server uses a random address that does not change.
6. bt\_gap\_connectable\_mode makes the device connectable and enables advertisement.

**Note**: A pointer to gap\_ops\_t instance is provided to this function call. This supplies the GAP callback functions connected\_cb and disconnected\_cb to be used when a connection or disconnection event occurs.

At this stage, it is suitable to accept the provisioning data from companion smartphone application and wait for the provisioning to complete.

|  |
| --- |
| int main(void)  {  . . .  . . .  start\_server();  //while(1), to be 'continued-in' for restarting prov after failure / timeout  // or to be 'breaken-out' for the prov success case  while(1)  {  os\_printf("Inno\_Ble\_WiFiProvisioning started\n");  while(!ssid\_provisioned || !pass\_provisioned)  os\_msleep(1000);  wifi\_main(ssid, pw);  . . .  }  . . .  } |

### BLE Connection/Disconnection Callbacks

At this point in the execution of server, it is advertising and ready to receive a connection from the client. When the client connects, the callback function connected\_cb will be called. In the callback, the GATT server needs to be linked to this GAP connection using bt\_gap\_server\_link\_add()with the following function call:

|  |
| --- |
| srv.gatt\_link = bt\_gap\_server\_link\_add(param->handle); |

The sample code provides details on how to obtain the argument required for this function call from the argument provided to the callback by casting hci\_event with bt\_hci\_evt\_le\_conn\_cmpl\_t and fetching its handle.

Similarly, the link is removed in the callback function that is called when the client disconnects disconnected\_cb, using bt\_gap\_server\_link\_remove():

|  |
| --- |
| bt\_gap\_server\_link\_remove(srv.gatt\_link); |

At disconnected\_cb, if either the SSID or Passphrase was not received, then the server is made connectable again. The bt\_gap\_connectable\_mode() makes server connectable again (will re-enable advertisement).

|  |
| --- |
| if(!ssid\_provisioned || !pass\_provisioned)  {  bt\_gap\_connectable\_mode(GAP\_CONNECTABLE\_MODE\_UNDIRECT, bt\_hci\_addr\_type\_random, 0, address\_zero, &gap\_ops);  } |

### BLE Characteristic Access Callback

While the client is connected to the server, it can read or write the custom characteristic based on characteristic’s properties. This results in the callback function associated with the characteristic being called.

When the write only characteristic UUID\_WIFI\_SSID\_16 is accessed, the callback associated when accessing this characteristic ssid\_provision\_cb() is called. BLE GATT Server receives this text messages from BLE Client and stores it as SSID. The bt\_att\_error\_t and ssid\_provision\_cb() callback functions are called when the custom characteristic is accessed.

|  |
| --- |
| static bt\_att\_error\_t ssid\_provision\_cb(uint8\_t bearer, bt\_gatt\_fcn\_t rw, uint8\_t \*length, uint8\_t offset, uint8\_t \*data)  {  if(offset != 0)  return BT\_ATT\_ERROR\_INVALID\_OFFSET;  ssid\_provisioned = 1;  memset(ssid, 0 , 32);  memcpy(ssid, data, \*length);  return BT\_ATT\_ERROR\_SUCCESS;  } |

When the write only characteristic UUID\_WIFI\_PASSCODE\_16 is accessed, the callback associated when accessing this characteristic pass\_provision\_cb() is called. BLE GATT server receives this text messages from BLE Client and stores it as Password.

bt\_att\_error\_t and pass\_provision\_cb() callback functions are called when the custom characteristic is accessed.

|  |
| --- |
| static bt\_att\_error\_t pass\_provision\_cb(uint8\_t bearer, bt\_gatt\_fcn\_t rw, uint8\_t \*length, uint8\_t offset, uint8\_t \*data)  {  if(offset != 0)  return BT\_ATT\_ERROR\_INVALID\_OFFSET;  pass\_provisioned = 1;  memset(pw, 0 , 32);  memcpy(pw, data, \*length);  return BT\_ATT\_ERROR\_SUCCESS;  } |

Once both the callbacks are received, both the flags pass\_provisioned and ssid\_provisioned are true, the program progresses to the next step. Talaria TWO then tries to connect to the provisioned SSID.

When the read only characteristic UUID\_WIFI\_STATUS\_16 is accessed, the callback associated when accessing this characteristic status\_provision\_cb() is called.

This is used by the smartphone enquire about the status of the connection attempt to AP.

A waiting, success, failure, or timeout status is passed as a string to the smartphone reading this characteristic, based on the present state. Few variables for keeping the states are also updated here.

bt\_att\_error\_t and status\_provision\_cb() callback functions are called when the custom characteristic is accessed.

|  |
| --- |
| static bt\_att\_error\_t status\_provision\_cb(uint8\_t bearer, bt\_uuid\_t \*uudid, bt\_gatt\_fcn\_t rw, uint8\_t \*length, uint16\_t offset, uint8\_t \*data)  /\* Writes to this characteristic not allowed \*/  if (rw != BT\_GATT\_FCN\_READ)  return BT\_ATT\_ERROR\_WRITE\_NOT\_PERMITTED;  if(offset != 0)  return BT\_ATT\_ERROR\_INVALID\_OFFSET; |

If status\_flag is 1, it implies that the IP address is received.

|  |
| --- |
| if(status\_flag)  {  memset(status, 0 , 16);  memcpy(status, STATUS\_SUCCESS, STATUS\_SUCCESS\_LENGTH);  \*length =STATUS\_SUCCESS\_LENGTH;  status\_sent = 1;  os\_printf("client reading status : success\n");  } |

If wcm\_return is 0 and the status\_flag is 0 it then waits for a notification.

|  |
| --- |
| else  {  if(wcm\_return == 0)  {  if(link\_down\_timeout == 0) /\* linkdown timer has not Timedout \*/  {  os\_printf("client reading status:waiting\n");  \*length =STATUS\_WAITING\_LENGTH;  } |

In this case, keep status\_sent to zero only and use timeout\_sent instead.

|  |
| --- |
| else /\* linkdown timer timedout \*/  {  /\* memcpy "timeout" as status, \*/  memset(status, 0 , 16);  memcpy(status, STATUS\_TIMEOUT, STATUS\_TIMEOUT\_LENGTH);  \*length =STATUS\_TIMEOUT\_LENGTH;  os\_printf("client reading status : timeout\n");  timeout\_sent = 1;  }  } |

If WCM returns any value, it leads to the client reading failure case.

|  |
| --- |
| else /\* non zero wcm\_return --> failure \*/  {  memset(status, 0 , 16);  memcpy(status, STATUS\_FAILURE, STATUS\_FAILURE\_LENGTH);  \*length =STATUS\_FAILURE\_LENGTH;  status\_sent = 1;  os\_printf("client reading status:failure\n");  }} |

memcpy() copies the status failure length and stores it in data.

|  |
| --- |
| memcpy(data, status, \*length);  return BT\_ATT\_ERROR\_SUCCESS; |

### Connecting to the Provisioned Wi-Fi network

To connect to a Wi-Fi network, wcm\_create(), wcm\_add\_network() and wcm\_auto\_connect()APIs from WCM are used. SSID and Password from section 7.3.6 are passed here.

wcm\_notify\_enable() is used to register notification callbacks for link-up, link-down and IP address changes. Based on these notification, connection success or timeout is decided.

A housekeeping structure for a timer is created for managing a timeout case in connection attempt.

**wifi\_control.c**

wcm\_create() creates the Wi-Fi network interface. wcm\_notify\_enable() enables the callbacks of link and IP address changes.

|  |
| --- |
| h = wcm\_create(NULL);  wcm\_notify\_enable(h, my\_wcm\_notify\_cb, NULL);  if( h == NULL ){  os\_printf(" failed.\n");  return -1;  } |

wcm\_add\_network() adds the network which must be added initially. It can fail due to the following reasons: already busy, no memory or poorly formatted password.

|  |
| --- |
| #ifdef WIFI\_CONNECT\_USE\_DEPRECATED\_WCM\_API  status = wcm\_add\_network(h, ssid, NULL, pw);  os\_printf("add network status: %d\n", status);  if(status != 0){  os\_printf("adding network Failed\n");  return status;  } |

struct network\_profile creates a Network Profile.

network\_profile\_new\_from\_file\_system() creates a Network Profile from a configuration file in the file system.

network\_profile\_new\_from\_ssid\_pw() creates a Network Profile using BOOT ARGS. In case of an error, the error messages are printed on console.

|  |
| --- |
| #else /\* WIFI\_CONNECT\_USE\_DEPRECATED\_WCM\_API \*/  /\*"/sys/nprofile.json"\*/  const char \*np\_conf\_path = os\_get\_boot\_arg\_str("np\_conf\_path")?: NULL;  struct network\_profile \*profile;  if (np\_conf\_path != NULL) {  status = network\_profile\_new\_from\_file\_system(&profile, np\_conf\_path);  } else {  status = network\_profile\_new\_from\_ssid\_pw(&profile, ssid, pw);  }  if (status < 0) {  pr\_err("could not create network profile %d\n", status);  return status;  }  status = wcm\_add\_network\_profile(h, profile);  if (status < 0) {  pr\_err("could not associate network profile to wcm %d\n", status);  return status;  } |

wcm\_auto\_connect() starts and stops the auto connection function.

|  |
| --- |
| #endif /\* WIFI\_CONNECT\_USE\_DEPRECATED\_WCM\_API \*/  os\_printf("added network successfully, will try connecting..\n");  status = wcm\_auto\_connect(h, 1);  os\_printf("connecting to network status: %d\n", status);  if(status != 0){  os\_printf("trying to connect to network Failed\n");  return status;  } |

This start connection attempts the timer which times out in TIMER\_TIMEOUT\_SEC\_IN\_MICRO seconds.

|  |
| --- |
| if(ptimer\_user\_data->timer\_running == 0)  {  ptimer\_user\_data->timeout = os\_systime()+(TIMER\_TIMEOUT\_SEC\_IN\_MICRO); |

This sets the timer callback and user data.

|  |
| --- |
| ptimer\_user\_data->timer\_id = os\_timer\_allocate(TIMER\_BASE\_US,  TIMER\_ANY, on\_timer\_event\_callback, ptimer\_user\_data); |

os\_timer\_set() function starts the timer for required timeout. The timeout value is in microseconds.

|  |
| --- |
| os\_timer\_set(ptimer\_user\_data->timer\_id, ptimer\_user\_data->timeout);  ptimer\_user\_data->timer\_running = 1;  connection\_attempt\_timeout = 0;  os\_printf ("\n connection attempt timer started. current time"  "in microseconds:[%u] \n", os\_systime());  } |

In the my\_wcm\_notify\_cb(), the timer is allocated using os\_timer\_allocate() and starts using os\_timer\_set() whenever a link-down is received and if the timer was not already running.

When the connection is later successful and a link-up occurs, this timer is cancelled if it was already running using os\_timer\_reset().

If the timer is not cancelled by link-up before the timeout occurs, then it is considered a connection trial time-out case. The timeout for this timer can be set using #define TIMER\_TIMEOUT\_SEC\_IN\_MICRO which by default is set to 8 seconds.

Few variables for keeping the states are also updated here:

|  |
| --- |
| static void my\_wcm\_notify\_cb(void \*ctx, struct os\_msg \*msg)  {  switch(msg->msg\_type)  {  case(WCM\_NOTIFY\_MSG\_LINK\_UP):  os\_printf("wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_LINK\_UP\n");  break;  case(WCM\_NOTIFY\_MSG\_LINK\_DOWN):  os\_printf("wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_LINK\_DOWN\n");  break;  case(WCM\_NOTIFY\_MSG\_ADDRESS):  os\_printf("wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_ADDRESS\n");  break;  case(WCM\_NOTIFY\_MSG\_DISCONNECT\_DONE):  os\_printf("wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_DISCONNECT\_DONE\n");  break;  case(WCM\_NOTIFY\_MSG\_CONNECTED):  os\_printf("wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_CONNECTED\n");  status\_flag = true;  /\* if active, then the connection\_attempt timer STOPS here \*/  if(ptimer\_user\_data->timer\_running == 1)  {  os\_timer\_reset(ptimer\_user\_data->timer\_id);  ptimer\_user\_data->timer\_running = 0;  os\_printf ("\n Cancelling the connection timeout timer. current time"  "in microseconds:[%u] \n", os\_systime());  /\* purpose of the timer is fulfilled, release it here \*/  os\_timer\_release(ptimer\_user\_data->timer\_id);  }  break;  default:  break;  }  os\_msg\_release(msg);  } |

If the timer is not cancelled before the programmed timeout, then a timer callback occurs where a flag is set indicating this scenario. Connection attempt timer's callback sets a flag to be used by att read cb and for the app\_main to behave accordingly.

|  |
| --- |
| static void on\_timer\_event\_callback(void \*user\_data)  {  os\_printf("\n Timeout Event occured.\n");  ptimer\_user\_data->timer\_running = 0;  /\* purpose of the timer is fulfilled, release it here \*/  os\_timer\_release(ptimer\_user\_data->timer\_id);  connection\_attempt\_timeout = 1;  } |

wcm\_delete\_network\_profile() deletes the currently active network profile that was previously added.

|  |
| --- |
| #ifndef WIFI\_CONNECT\_USE\_DEPRECATED\_WCM\_API  wcm\_delete\_network\_profile(h, NULL);  #endif |

This cleans the ptimer\_user\_data.

|  |
| --- |
| if(ptimer\_user\_data!= NULL)  {  os\_free(ptimer\_user\_data);  ptimer\_user\_data = NULL;  } |

wcm\_destroy() function shuts down and cleans up a Wi-Fi Connection Manager interface.

|  |
| --- |
| wcm\_destroy(h); |

**main.c**

In main(), after calling wifi\_main(), the return from this function and the state changes from callbacks on\_timer\_event\_callback() and my\_wcm\_notify\_cb() are used to decide if the connection attempt was successful, failed or timed-out.

If the attempt results in error, failure or timeout then the relevant status is sent to the smartphone app when it enquires, and the provisioning loop starts again.

Once the Wi-Fi connection is successful, the BT GATT and GAP resources are destroyed using common\_server\_destroy() and bt\_gap\_destroy(), and T2 stops advertising for BLE connection.

In every step when status change happens, it is made sure that the status has been read by smartphone app using status\_provision\_cb() before going to next state.

|  |
| --- |
| int main(void)  status\_flag = false;  memset(status, 0 , 16);  memcpy(status, STATUS\_WAITING, STATUS\_WAITING\_LENGTH); |

bt\_gap\_init() initiates the Bluetooth GAP service.

|  |
| --- |
| bt\_gap\_init(); |

The common server implements the following services that will be added to the server. The services are Generic Access, Generic Attribute and Device information.

|  |
| --- |
| common\_server\_create("Inno\_Ble\_WiFiProvisioning", 0, "Innophase Inc"); |

As we are restarting provisioning, reset the housekeeping state variables and status message to default value waiting.

|  |
| --- |
| memset(status, 0 , 16);  memcpy(status, STATUS\_WAITING, STATUS\_WAITING\_LENGTH);  status\_sent = 0;  timeout\_sent = 0;  connection\_attempt\_timeout = 0;  wcm\_return = 0;  continue; |

If connection\_attempt\_timeout happens before connection success, then continue to while(1) loop after resetting the flags: ssid\_provisioned, pass\_provisioned and executing wifi\_destroy. In this case, the status message sent is STATUS\_TIMEOUT and the flag timeout\_sent is used to confirm that the status STATUS\_TIMEOUT has been sent to the client.

|  |
| --- |
| if(connection\_attempt\_timeout == 1)  {  /\*connection attempt timed-out. also, status\_sent is 0 in timeout case \*/  os\_printf("int main -- WiFi Connection not succesfull"  " because connection attempt timed-out \n");  /\* make sure status timeout is sent, before starting prov again \*/  while(!timeout\_sent)  {  os\_msleep(1000);  }  wifi\_destroy(0); |

Since provisioning is restarting, reset the housekeeping state variables and status message to default value waiting. Continue to the provisioning loop after resetting ssid\_provisioned, pass\_provisioned and housekeeping state variables.

|  |
| --- |
| ssid\_provisioned = 0;  pass\_provisioned = 0;  connection\_attempt\_timeout = 0;  wcm\_return = 0;  memset(status, 0 , 16);  memcpy(status, STATUS\_WAITING, STATUS\_WAITING\_LENGTH); |

This starts provisioning again, hence reset the remaining housekeeping state variables.

|  |
| --- |
| timeout\_sent = 0;  status\_sent = 0 |

This function destroys the common server and the Bluetooth GAP service.

|  |
| --- |
| common\_server\_destroy();  bt\_gap\_destroy(); |

my\_app\_init() function starts the app here.

|  |
| --- |
| my\_app\_init(); |

After successful connection, another application thread is created for example purposes where the application logic can be run.

|  |
| --- |
| static void my\_app\_init()  {  my\_app\_thread = os\_create\_thread("my\_app\_thread", (void \*) my\_app\_thread\_func,  NULL, MY\_APP\_THREAD\_PRIO, MY\_APP\_THREAD\_STACK\_SIZE);  if( my\_app\_thread == NULL ){  os\_printf(" thread creation failed\n");  return;  }  os\_join\_thread(my\_app\_thread); } |

# Running the Application

## Program Talaria TWO

Program ble\_provisioning.elf(sdk\_x.y\apps\ble\_provisioning\bin) using the Download tool(sdk\_x.y\pc\_tools\Download\_Tool) provided with InnoPhase Talaria TWO SDK.

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down
   2. ELF Input: Load the ble\_provisioning.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y\pc\_tools\Download\_Tool\doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

## Using InnoPhase Talaria TWO Smart Home Android Application

To test this sample application, we will need to use the companion Innophase T2 Smart Home Android application on an android device.

1. To install, open the provided .apk file (sdk\_x.y\apps\ble\_provisioning\mobile\_app) in the android directory or build the android project using Android Studio.
2. To connect to the Talaria TWO BLE Server, wait for the application to complete the scanning and look for Inno\_Ble\_WiFiProvisioning and click on it.



Figure : Scanning for Talaria TWO BLE Server for Wi-Fi Provisioning

Android phone connects as a BLE Client to Talaria TWO device at this stage.

1. Android application scans for the nearby available Wi-Fi networks and displays them in a list view.

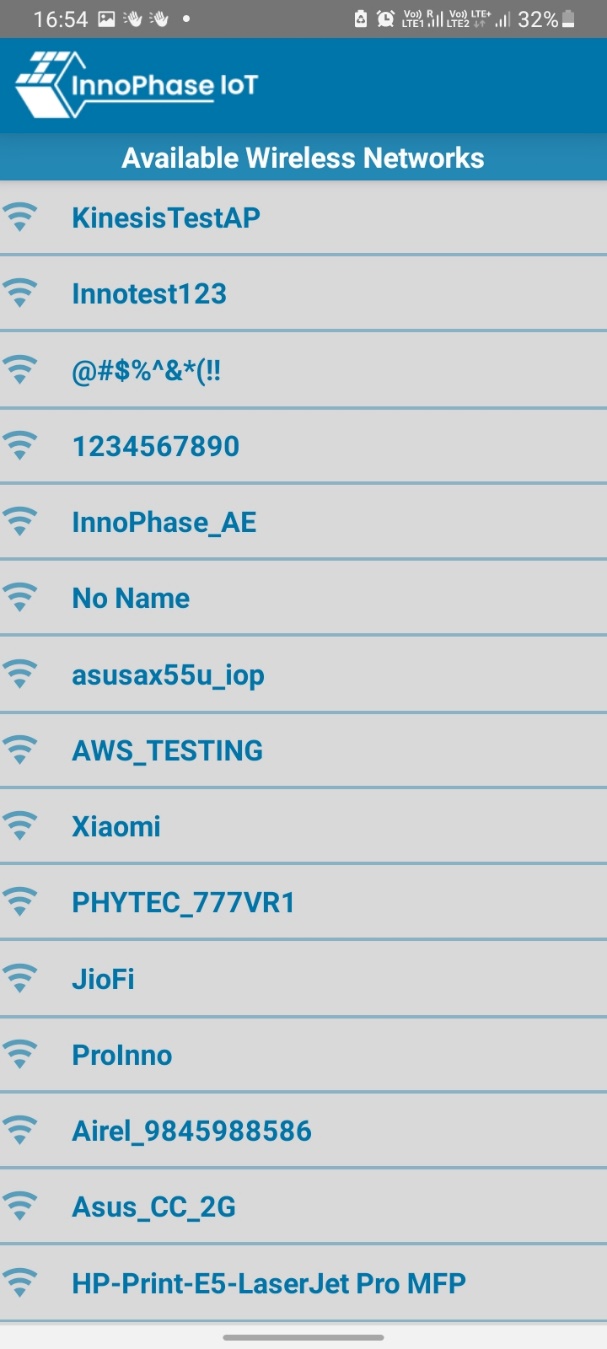


Figure : Available Wi-Fi networks as scanned by Android Phone

1. Select the SSID of the AP you want to connect to and provide the passphrase.

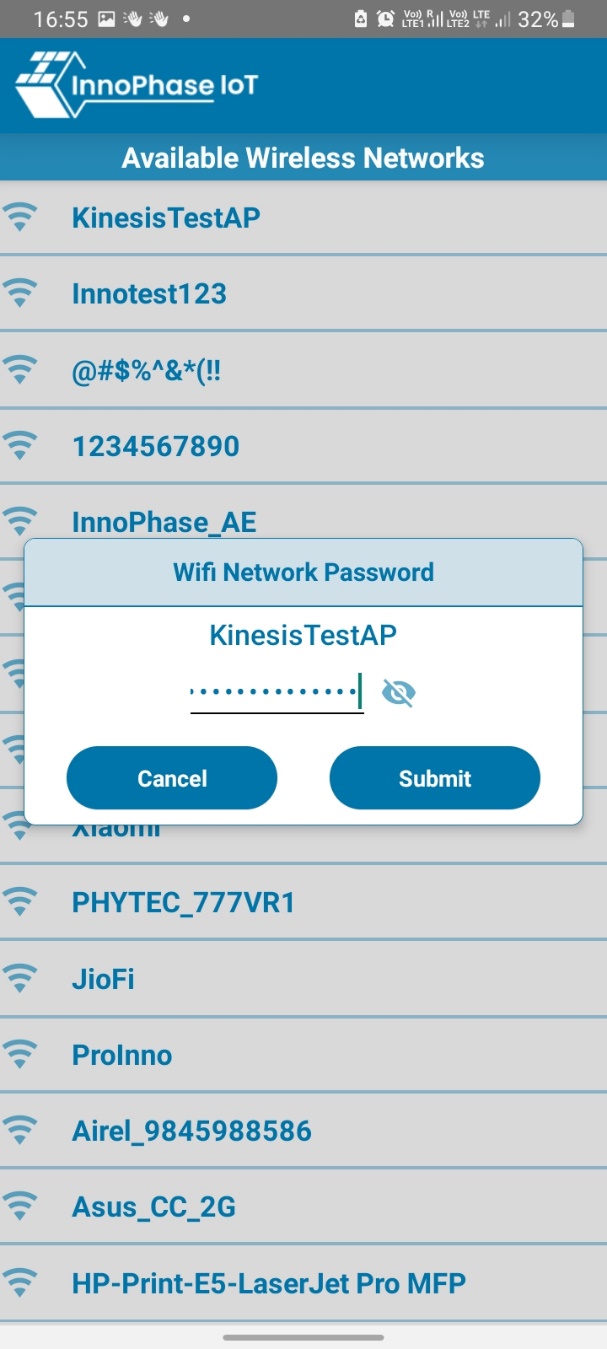


Figure : Providing the passphrase

1. Once the passphrase is entered, click on Done. If the provided passphrase is correct, connection will be established successfully. If not, an error message is shown.

Graphical user interface, application, PowerPoint

Description automatically generated

Figure : Connecting successful

Graphical user interface, application

Description automatically generated

Figure : Error in connection

1. On establishing the connection successfully, the android application should transfer the Wi-Fi credentials using custom GATT Service and Characteristics we created.

Shape

Description automatically generated with low confidence

Figure : Connection successful

# Expected Output

Talaria TWO will try to connect to the provisioned network and provide the following console output:

|  |
| --- |
| UART:NWWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-33c6d3fa5 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-33c6d3fa5 $  Inno\_Ble\_WiFiProvisioning started  [23.887,073] BT connect[0]: ia:79:49:1f:f3:24:87 aa:05:04:03:02:01:00 phy2:0/0 phyC:00  Client connected  client reading status:waiting  WiFi Details SSID: InnoPhase\_AE\_AP, PASSWORD: innophaseae  addr e0:69:3a:00:15:c2  Connecting to WiFi...  added network successfully, will try connecting..  connecting to network status: 0  connection attempt timer started. current timein microseconds:[25245506]  [25.902,728] CONNECT:98:da:c4:73:b7:76 Channel:2 rssi:-46 dBm  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_LINK\_UP  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_ADDRESS  [25.953,426] MYIP 192.168.0.164  [25.953,706] IPv6 [fe80::e269:3aff:fe00:15c2]-link  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_CONNECTED  Cancelling the connection timeout timer. current timein microseconds:[25953775]  client reading status : success  status sent to phone app, now calling common\_server\_destroy and bt\_gap\_destroy  starting app\_thread |