Talaria TWO™(INP2045)

Low Power Multi-Protocol Wireless Platform SoC

IEEE 802.11 b/g/n, BLE 5.0

Application Note

BLE Provisioning

Release: 07-19-2021

InnoPhase, Inc.

6815 Flanders Drive

San Diego, CA 92121

innophaseinc.com

Revision History

Version Number	Date	Comments
1.0	07-04-2020	First release
1.1	10-14-2020	Enhanced the application code and APK for error handling
2.0	04-23-2021	Enhanced application outputs to print SDK version
3.0	07-19-2021	Updated for SDK 2.3 release

Contents

1 Figures

2 Terms & Definitions

AP	Access Point
API	Application Programming Interface
BLE	Bluetooth Low Energy
GAP	Generic Access Profile
GATT	Generic Attribute Profile
SSID	Service Set Identifier
UUID	Universally Unique Identifier
WCM	Wi-Fi Connection Manager
WPA	Wi-Fi Protected Access

3 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.

4 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.

5 APIs used for this Application

5.1 BLE APIs used

5.1.1 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.

5.1.2 common_server_create()

Creates a server with the name, manufacturer name and appearance passed, and creates and adds below services to the created server:

1. Generic Access
2. Generic Attribute
3. Device Information Services

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

```
void common_server_create(char *name, uint16_t appearance, char

*manufacturer_name);
```

5.1.3 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();
```

5.1.4 bt_gatt_create_service_128()

Create a service declaration from a 128-bits UUID given as parameter and returns the pointer to the GATT service.

```
struct gatt_service * bt_gatt_create_service_128(uint128_t uuid128)
```


5.1.5 bt_gatt_add_char_16()

Adds a characteristic with a 16-bit UUID to a created service.

It takes permission, properties and an access callback function as input which is called by stack when this characteristic is accessed.

```
struct gatt_char * bt_gatt_add_char_16(struct gatt_service *s, uint16_t

uuid16, bt_srv_fcn_t fcn, uint8_t permission, uint8_t property)
```


The characteristic property can be any from the following list:

```
/**
 * GATT characteristic properties
 */
#define GATT_CHAR_PROP_BIT_BROADCAST    (1<<0)
#define GATT_CHAR_PROP_BIT_READ         (1<<1)
#define GATT_CHAR_PROP_BIT_WRITE_NO_RSP (1<<2)
#define GATT_CHAR_PROP_BIT_WRITE        (1<<3)
#define GATT_CHAR_PROP_BIT_NOTIFY       (1<<4)
#define GATT_CHAR_PROP_BIT_INDICATE     (1<<5)
#define GATT_CHAR_PROP_BIT_WRITE_SIGNED (1<<6)
#define GATT_CHAR_PROP_BIT_EXT_PROP     (1<<7)
#define GATT_CHAR_PROP_R    (GATT_CHAR_PROP_BIT_READ)
#define GATT_CHAR_PROP_W    (GATT_CHAR_PROP_BIT_WRITE)
#define GATT_CHAR_PROP_WN   (GATT_CHAR_PROP_BIT_WRITE_NO_RSP)
#define GATT_CHAR_PROP_WNS  (GATT_CHAR_PROP_WN |
GATT_CHAR_PROP_BIT_WRITE_SIGNED)
#define GATT_CHAR_PROP_WS   (GATT_CHAR_PROP_W  |
GATT_CHAR_PROP_BIT_WRITE_SIGNED)
#define GATT_CHAR_PROP_RW   (GATT_CHAR_PROP_R | GATT_CHAR_PROP_W)
#define GATT_CHAR_PROP_RWN  (GATT_CHAR_PROP_R | GATT_CHAR_PROP_WN)
#define GATT_CHAR_PROP_RWNS (GATT_CHAR_PROP_R | GATT_CHAR_PROP_WNS)
#define GATT_CHAR_PROP_RWS  (GATT_CHAR_PROP_R | GATT_CHAR_PROP_WS)
#define GATT_CHAR_PROP_N    (GATT_CHAR_PROP_BIT_NOTIFY)
#define GATT_CHAR_PROP_I    (GATT_CHAR_PROP_BIT_INDICATE)
```


5.1.6 bt_gatt_add_service()

Adds a created service to the local server list. All `includes`, `characteristics` and `descriptors` should have been added to the created service before the service is added to the server.

```
void bt_gatt_add_service(struct gatt_service *s)
```

5.1.7 bt_gap_cfg_adv()

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.

```
bt_gap_error_t bt_gap_cfg_adv(const uint16_t adv_fast_period, const

uint16_t adv_slow_period, const uint16_t adv_fast_int, const uint16_t

adv_slow_int, const int8_t adv_tx_power , const uint8_t adv_ch_map)
```

The API takes the following parameters as inputs:

1. `adv_fast_period(ms)`: for this period, fast advertising is attempted every `adv_fast_int` interval. Once this period is completed, slow advertising is attempted every `adv_slow_int` interval. Default value of this parameter is 0, representing period infinity, which means fast advertising will be attempted forever once started.
2. `adv_slow_period (ms)`: for this period, slow advertising is attempted every `adv_slow_int` interval. Once this period is completed, advertising is disabled. Default value of this parameter is 0, representing period infinity, which means slow advertising will be attempted forever once started.
3. `adv_fast_int` in 625µs units: This sets the interval between two fast advertisements. Range: 0x0020 to 0x4000 (default: 200).

   Which implies, when this interval is represented in decimal, the range is between 20,000µs (20ms) to 10,240,000µs (10,240ms) configurable in the steps of 625µs. Default in decimal being 125,000µs, which is, every 125ms, 8 times per second.
4. `adv_slow_int` in 625µs units: This sets the interval between two slow advertisements. Range: 0x0020 to 0x4000 (default: 1,600).

Which implies, when this interval is represented in decimal, the range is between 20,000µs (20ms) to 10,240,000µs (10,240ms) configurable in steps of 625µs. Default in decimal being 1,000,000µs, which is, every 1,000ms once per second.

5. `adv_tx_power` in dBm, range: -127 to 10, and 127 (127=no preference) (default: 127)
6. `adv_ch_map` Channel map used: bit0=ch37, bit1=ch38, bit2=ch39 (default: 0x7)

The API returns error code from `bt_gap_error_t`.

5.1.8 bt_gap_connectable_mode()

Sets the device in desired connectable mode.

```
bt_gap_error_t bt_gap_connectable_mode(const gap_connectable_mode_t
mode, const bt_hci_addr_type_t
own_type, const bt_hci_addr_type_t peer_type, const bt_address_t
peer_address, const gap_ops_t *ops)
```


The connection mode can be any from the following list:

```
typedef enum {

    /** Disable connectable mode */

    GAP_CONNECTABLE_MODE_DISABLE  = 0,

    /** Do not allow a connection to be established */

    GAP_CONNECTABLE_MODE_NON      = 1,

    /** Accept a connection request from a known peer device */

    GAP_CONNECTABLE_MODE_DIRECT   = 2,

    /** Accept a connection request from a any device */

    GAP_CONNECTABLE_MODE_UNDIRECT = 3,

} gap_connectable_mode_t;
```

Other input parameters to this API are:

1. `own_type`: Own address type: 0=public, 1=random, 2=resolvable (or public if no local IRK), 3=resolvable (or random if no local IRK)
2. `peer_type`: Peer address type: 0=public (device or identity), 1=random (device or identity)
3. `peer_address`: Peer address
4. `ops`: GAP callback functions. Ex: connection and disconnection callback

5.1.9 bt_gap_server_link_add()

Used to add a GATT server to the GAP connection.

```
struct gatt_srv_link * bt_gap_server_link_add(const uint8_t handle)
```

It takes connection handle as input and returns pointer to `gatt_srv_link`.

5.1.10 bt_gap_server_link_remove()

Used to remove GATT server from the GAP connection.

```
void bt_gap_server_link_remove(const struct gatt_srv_link *link)
```

It takes pointer to `gatt_srv_link` to be removed as input.

5.2 Wi-Fi Connection Manager APIs used

5.2.1 wcm_create()

To use WCM, the first API to be called is `wcm_create()`, where initializations of different components are taken care of. This returns a `wcm_handle`.

```
struct wcm_handle * wcm_create(const uint8_t *hwaddr)
```

`*hwaddr` holds the user defined `hw` address. If it is defined as NULL, WCM uses a random `hw` address. Returns a pointer to `wcm_handle`.

5.2.2 wcm_connect()

Synchronously connect to a Wi-Fi network.

```
int wcm_connect(struct wcm_handle *h, const char *ssid, const char

*passphrase)
```

Parameters passed are:

1. `struct wcm_handle *h`: Pointer to `wcm_handle`
2. `const char *ssid`: Pointer to string with the SSID of the desired network
3. `const char *passphrase`: The passphrase used to generate the shared secret which is in-turn used to encrypt the traffic on the network. For a network secured using WPA/WPA2, either a passphrase of 8 to 63 characters or a HEX formatted key of 64 characters can be used.

Returns zero on success, negative error code in case of an error as listed:

1. EBUSY -- A network is already configured
2. ENOMEM -- Not enough memory
3. EIBADF -- Badly formatted passphrase

Note: w`wcm_connect()` API is used to synchronously connect to a Wi-Fi network, which means, the thread calling this API hangs until a successful connection has been established. It can even hang indefinitely if the network is not found or if the passphrase is incorrect. For dynamic usage it is recommended to use the `wcm_add_network()` function instead which is an asynchronous way to achieve same result.

6 Code Walkthrough

6.1 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.

6.2 Sample Code Walkthrough

6.2.1 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.

6.2.2 Adding Common GATT Server Functionality

The server uses the API `common_server_create()` to add the common server functionality. This adds the Generic Access, Generic Attribute, and Device Information services to the server.

The server is given the name `Inno_BLEWiFiProvisioning` with a manufacturer name of `Innophase Inc`.

```
common_server_create("Inno_BLEWiFiProvisioning", 0, "Innophase Inc");
```

6.2.3 Adding Custom GATT Service & Characteristic

The server's custom service is created, and characteristics are added to it:

```
/* UUIDs of our custom characteristics */

#define UUID_WIFI_SSID_16     0xAB34

#define UUID_WIFI_PASSCODE_16 0xCD34

#define UUID_WIFI_STATUS_16   0xEF34

. . .

. . .


/* Creates a "custom server" i.e. adds our custom service and

characteristic */

static void custom_server_create(void)

{

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);
}
```

The `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.

6.2.4 Starting BLE GATT Server

Once the server's services and characteristics are set up, it is started in the `start_server` function:

```
/* GAP option object to be passed to GAP functions */
static const gap_ops_t gap_ops = {
    .connected_cb    = connected_cb,
    .disconnected_cb = disconnected_cb,
    .discovery_cb    = NULL,
};


. . .
. . .

static void start_server(void)
{
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);

    /* Set our BLE address */
    bt_gap_addr_set(bt_hci_addr_type_random, SERVER_ADDR);
```



```
    /* Set our BLE address */
    bt_gap_addr_set(bt_hci_addr_type_random, SERVER_ADDR);


    /* Make server connectable (will enable 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 our device via Bluetooth, we must start advertising and make our device connectable.

Here, `bt_gap_cfg_adv` 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.

2. `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.

3. `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.

4. `adv_slow_int` is set to 1,600, which entails (1,600*625µs) = 1,000,000µs = every second once will be the interval of slow advertising.

`bt_gap_set_adv_data` sets the advertisement data. It is for legacy advertisement.

`bt_gap_addr_set` sets our BLE address and address type. The sample server uses a `random` address that does not change. `bt_gap_connectable_mode` makes the device connectable and will enable 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, we are ready 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);
. . .
  }
. . .
}
```

6.2.5 BLE Connection/Disconnection Callbacks

At this point in the execution of the 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.

```
if(!ssid_provisioned || !pass_provisioned)

{

      // Make server connectable again (will re-enable advertisement)

      bt_gap_connectable_mode(GAP_CONNECTABLE_MODE_UNDIRECT,

bt_hci_addr_type_random, 0, address_zero, &gap_ops);

}
```

6.2.6 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.

```
/* Callback called when our 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.

```
/* Callback called when our 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.

```
/* Callback called when our 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;

    /* status_flag is 1, means we got the IP */
    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");
    }
    else
    {
        /* wcm_return is 0 AND status_flag is 0 --> wait for
notifications */
        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;
            }
            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");
                /* in this case, keep status_sent to zero only and
                   use timeout_sent instead */
                timeout_sent = 1;
            }
        }
        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(data, status, *length);


    return BT_ATT_ERROR_SUCCESS;

}
```

6.2.7 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 the WCM are used. SSID and Password from section 6.2.6 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.

main.c

```
. . .
#define TIMER_TIMEOUT_SEC_IN_MICRO 8000000
typedef struct timer_user_data_t
{
  //unsigned int timer_created_at;
  unsigned int timeout;
  unsigned int timer_running;
  os_timer_id_t timer_id;
}timer_user_data;

static timer_user_data *ptimer_user_data;
. . .
. . .

int wifi_main(char *ssid, char *pw)
{

    /*creating a timer*/
    /*allocating timer user data*/
```

```
    ptimer_user_data =  os_alloc(sizeof(timer_user_data));
    ptimer_user_data->timer_running = 0;
. . .
. . .
      int status;

    os_printf("\n\rWiFi Details  SSID: %s, PASSWORD: %s\n\r", ssid, pw);

    h = wcm_create(NULL);
    wcm_notify_enable(h, my_wcm_notify_cb, NULL);
    if( h == NULL ){
        os_printf(" failed.\n");
    return -1;
    }

    os_printf("Connecting to WiFi...\n");
    /* async connect to a WiFi network */

    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");
        /* can fail due to, already busy, no memory,
          wor badly formatted password */
        return status;
```

```
    }

    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");
        /* can fail due to, already busy, no memory */
        return status;
    }


#if 0
    /* RELOCATE below itmes -- will need to change place in new async
way */

    /* bad when timer was not created in linkdown? if yes,
      then release in linkup etc */
    //os_timer_release(ptimer_user_data->timer_id);

    os_free(ptimer_user_data);
#endif

    return status;
}
```


In the `my_wcm_notify_cb()`, the timer is allocated using `os_timer_allocate()` and started 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");
        /* if timer active, then the linkdown 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 Timeout Timer. \
                 current time:[%u] \n", os_systime());
        }
            break;

        case(WCM_NOTIFY_MSG_LINK_DOWN):
        os_printf("wcm_notify_cb to App Layer -
WCM_NOTIFY_MSG_LINK_DOWN\n");

        /* a timer starts on linkdown (IF already NOT running),
          timeout 5 seconds (#define TIMER_TIMEOUT_SEC_IN_MICRO) */
        if(ptimer_user_data->timer_running == 0)
        {
            //ptimer_user_data->timer_created_at = os_systime();
```

```
            ptimer_user_data->timeout = os_systime()+
                 (TIMER_TIMEOUT_SEC_IN_MICRO);

            /*setting 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);

            /*starting the timer for required timeout.
                 timeout time is in microseconds*/
            os_timer_set(ptimer_user_data->timer_id,
                  ptimer_user_data->timeout);
            ptimer_user_data->timer_running = 1;
            os_printf ("\n Linkdown Timeout Timer started. \
                  current time:[%u] \n", os_systime());
        }

            break;

        case(WCM_NOTIFY_MSG_ADDRESS):
        os_printf("wcm_notify_cb to App Layer -
WCM_NOTIFY_MSG_ADDRESS\n");
        status_flag = true;
        /* if AP goes OFF after giving IP so linkdown happens and the IP
           becomes 0, then also status is becoming 1 -- check later */

        /* if active, then the linkdown timer STOPS here (mostly, timer
          will never be active here, still) */
```

```
        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 Timeout Timer. \
                     current time:[%u] \n", os_systime());
        }
            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.

```
//linkdown timer's callback just sets a flag, to be used by att read cb
and app_main to behave accordingly
static void on_timer_event_callback(void *user_data)
{
    os_printf("\n Timeout Event occured.\n");
    link_down_timeout =1;
}
```

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, its made sure that status has been read by smartphone app using `status_provision_cb()` before going to next state.

```
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_printf("debug 1 loop \n");
            os_msleep(1000);
        }

        wcm_return = wifi_main(ssid,pw);
        if(wcm_return != 0)
        {
            os_printf("main -- WiFi Connection Failed due to \
                  WCM returning error \n");
            /* continue to provisioning loop again resetting
              ssid_provisioned pass_provisioned. */
```



```
            ssid_provisioned = 0;

            pass_provisioned = 0;

            /* make sure status 'failure' is sent, before starting prov
again */

            while(!status_sent)

            {

                os_msleep(1000);

                //os_printf("debug 2 loop \n");

            }

            wifi_destroy(0);


            /* As we are restarting provisioning, reset the housekeeping
and

               status mssg to default value 'waiting' */


            memset(status, 0 , 16);

            memcpy(status, STATUS_WAITING, STATUS_WAITING_LENGTH);

            status_sent = 0;

            timeout_sent = 0;

            link_down_timeout = 0;

            wcm_return = 0;

            //os_printf("debug 3 continue \n");

            continue;

        }

        else

        {

            while(!status_sent)

            {
```

```
                /* IF timeout in linkdown happens due to TIMER, we
continue
                to while(1) loop after resetting ssid_provisioned
                pass_provisioned and doing wifi_destroy etc */
                /* status mssg sent is 'timeout' and timeout_sent is
used
                to confirm that status 'timeout' has been read by client
*/
                /* linkdown Timeout happened. also status_sent is 0 in
                timeout case */
                if(link_down_timeout == 1)
                {
                    os_printf("int main -- WiFi Connection not
succesfull \
                              due to LINK DOWN timeout scenario \n");

                    /* make sure status timeout is sent, before starting
                      prov again */
                    while(!timeout_sent)
                    {
                        os_msleep(1000);
                        //os_printf("debug 4 loop \n");
                    }
                    wifi_destroy(0);

                    /* as we are restarting provisioning, reset the
housekeeping
                      and status mssg to default value 'waiting' */
```

```
                    /* can continue to provisioning loop again resetting
                      ssid_provisioned pass_provisioned and housekeeping
                      variables */
                    ssid_provisioned = 0;
                    pass_provisioned = 0;
                    link_down_timeout = 0;
                    wcm_return = 0;

                    memset(status, 0 , 16);
                    memcpy(status, STATUS_WAITING,
STATUS_WAITING_LENGTH);

                    /* break from while (!status_sent), and later use
timeout_sent
                    to either break from or continue to provisioning
while(1) */
                    status_sent = 1;

                    continue;
                }
                else /* if linkdown NOT timedout */
                {
                    os_msleep(1000);
                    //os_printf("debug 5 loop \n");
                }
            }
```



```
            if(timeout_sent == 1)
            {
                /* starting provisioning again, so reset remaining
                housekeeping  */
                timeout_sent = 0;
                status_sent = 0;

                //os_printf("debug 6 continue \n");
                continue;
            }
            else
            {
                //os_printf("debug 7 break \n");
                break;
            }
        }
    }/* while(1) ends here */

    os_printf("status sent to phone app, now calling
common_server_destroy \
             and bt_gap_destroy\n");
    common_server_destroy();
    bt_gap_destroy();

    /* start your app here */
    my_app_init();

    while(true)
```

```
        os_msleep(1000);


    return 0;

}
```

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); }
```


7 Running the Application

7.1 Program Talaria TWO

Program the Talaria TWO development board using the Download tool.

Launch the Download tool provided with InnoPhase Talaria TWO SDK. In the GUI window, select the appropriate EVK from the drop-down and load `ble_provisioning.elf`. Prog RAM or Prog Flash as per requirement.

For details on using the Download tool, refer to the document: UG_Download_Tool.pdf.

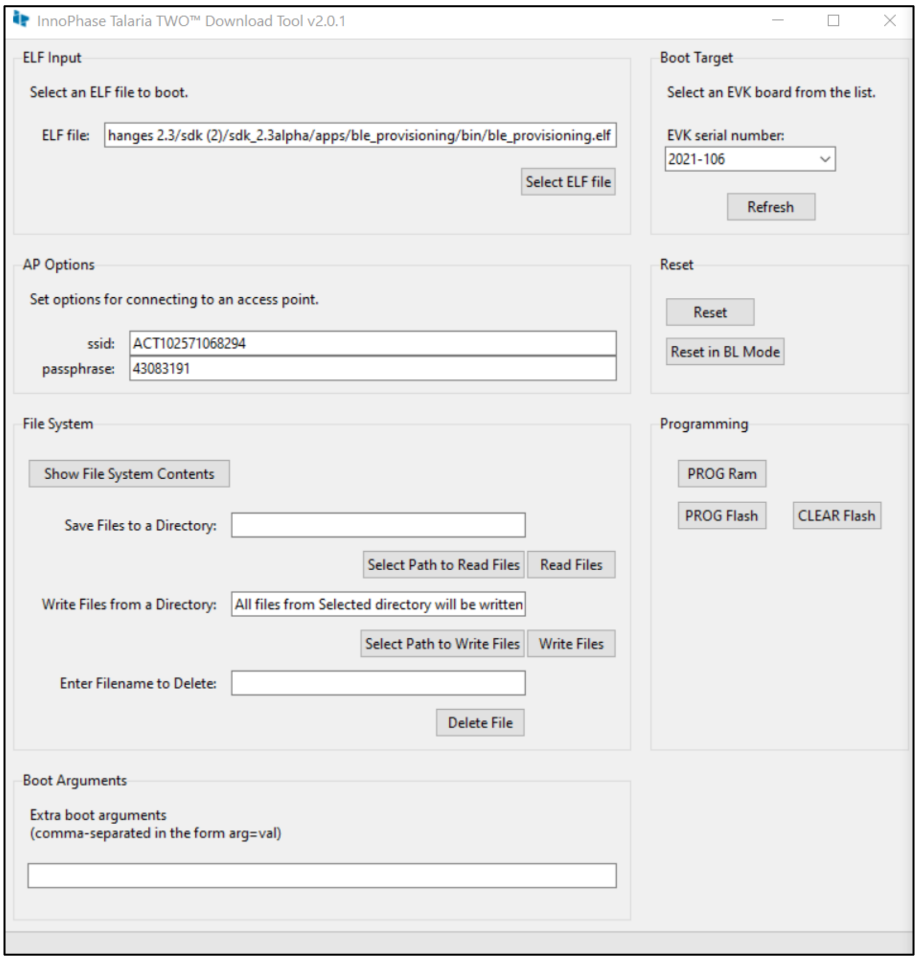
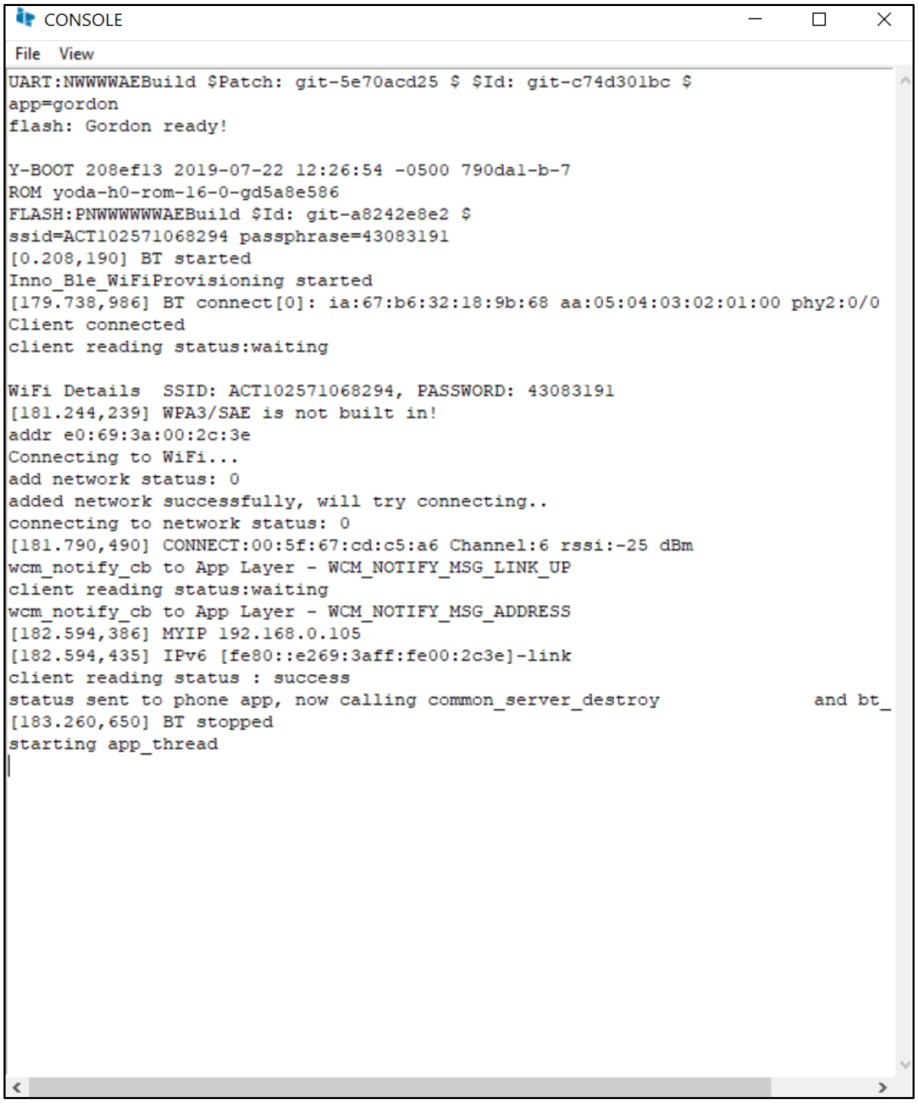


Figure 1: Download tool GUI



```
CONSOLE
File View
UART:NWWWWAEBuild $Patch: git-5e70acd25 $ $Id: git-c74d301bc $
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:PNWWWWWWAEBuild $Id: git-a8242e8e2 $
ssid=ACT102571068294 passphrase=43083191
[0.208,190] BT started
Inno_Ble_WiFiProvisioning started
[179.738,986] BT connect[0]: ia:67:b6:32:18:9b:68 aa:05:04:03:02:01:00 phy2:0/0
Client connected
client reading status:waiting

WiFi Details  SSID: ACT102571068294, PASSWORD: 43083191
[181.244,239] WPA3/SAE is not built in!
addr e0:69:3a:00:2c:3e
Connecting to WiFi...
add network status: 0
added network successfully, will try connecting..
connecting to network status: 0
[181.790,490] CONNECT:00:5f:67:cd:c5:a6 Channel:6 rssi:-25 dBm
wcm_notify_cb to App Layer - WCM_NOTIFY_MSG_LINK_UP
client reading status:waiting
wcm_notify_cb to App Layer - WCM_NOTIFY_MSG_ADDRESS
[182.594,386] MYIP 192.168.0.105
[182.594,435] IPv6 [fe80::e269:3aff:fe00:2c3e]-link
client reading status : success
status sent to phone app, now calling common_server_destroy           and bt_
[183.260,650] BT stopped
starting app_thread
```

Figure 2: Download tool output

7.2 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` and an android device.

1. To install, open the provided `.apk` file 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.

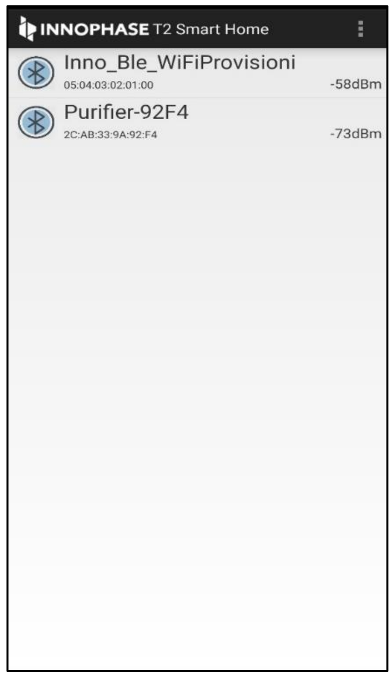


Figure 3: Scanning for Talaria TWO BLE Server for Wi-Fi Provisioning

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

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

Figure 4: Available Wi-Fi networks as scanned by Android Phone

4. Select the SSID of the AP you want to connect to. A passphrase needs to be provided for the SSID.

Figure 5: Providing the passphrase

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

Figure 6: Connecting successful

Figure 7: Error in connection

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

Figure 8: Connection successful

8 Expected Output

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

```
UART:NWWWWAEBuild $Patch: git-5e70acd25 $ $Id: git-c74d301bc $
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:PNWWWWWWAEBuild $Id: git-a8242e8e2 $
ssid=ACT102571068294 passphrase=43083191
[0.208,190] BT started
Inno_Ble_WiFiProvisioning started
[179.738,986] BT connect[0]: ia:67:b6:32:18:9b:68 aa:05:04:03:02:01:00 phy2:0/0
phyC:00
Client connected
client reading status:waiting


WiFi Details  SSID: ACT102571068294, PASSWORD: 43083191

[181.244,239] WPA3/SAE is not built in!
addr e0:69:3a:00:2c:3e
Connecting to WiFi...
add network status: 0
added network successfully, will try connecting..
connecting to network status: 0
[181.790,490] CONNECT:00:5f:67:cd:c5:a6 Channel:6 rssi:-25 dBm
wcm_notify_cb to App Layer - WCM_NOTIFY_MSG_LINK_UP
```

```
client reading status:waiting

wcm_notify_cb to App Layer - WCM_NOTIFY_MSG_ADDRESS

[182.594,386] MYIP 192.168.0.105

[182.594,435] IPv6 [fe80::e269:3aff:fe00:2c3e]-link

client reading status : success

status sent to phone app, now calling common_server_destroy and
bt_gap_destroy

[183.260,650] BT stopped

starting app_thread
```

9 Support

1. Sales Support: Contact an InnoPhase sales representative via email – sales@innophaseinc.com
2. Technical Support:
    a. Visit: https://innophaseinc.com/contact/
    b. Also Visit: https://innophaseinc.com/talaria-two-modules
    c. Contact: support@innophaseinc.com

InnoPhase is working diligently to provide outstanding support to all customers.

10 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, InnoPhase Incorporated does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and assumes no liability associated with the use of such information. InnoPhase Incorporated takes no responsibility for the content in this document if provided by an information source outside of InnoPhase Incorporated.

InnoPhase Incorporated disclaims liability for any indirect, incidental, punitive, special or consequential damages associated with the use of this document, applications and any products associated with information in this document, whether or not such damages are based on tort (including negligence), warranty, including warranty of merchantability, warranty of fitness for a particular purpose, breach of contract or any other legal theory. Further, InnoPhase Incorporated accepts no liability and makes no warranty, express or implied, for any assistance given with respect to any applications described herein or customer product design, or the application or use by any customer's third-party customer(s).

Notwithstanding any damages that a customer might incur for any reason whatsoever, InnoPhase Incorporated' aggregate and cumulative liability for the products described herein shall be limited in accordance with the Terms and Conditions of identified in the commercial sale documentation for such InnoPhase Incorporated products.

Right to make changes — InnoPhase Incorporated reserves the right to make changes to information published in this document, including, without limitation, changes to any specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — InnoPhase Incorporated products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an InnoPhase Incorporated product can reasonably be expected to result in personal injury, death or severe property or environmental damage. InnoPhase Incorporated and its suppliers accept no liability for inclusion and/or use of InnoPhase Incorporated products in such equipment or applications and such inclusion and/or use is at the customer's own risk.

All trademarks, trade names and registered trademarks mentioned in this document are property of their respective owners and are hereby acknowledged.

