This application note describes the accompanying example code in which Talaria TWO receives a text message from a connected BLE client and publishes that text message to a MQTT Broker.

# MQ Telemetry Transport Protocol

MQTT is a messaging protocol based on publish-subscribe pattern. It works on top of the TCP/IP protocol and is used in internet of things.

Publish-subscribe paradigm is event-driven, and messages are pushed to the clients. It requires an additional central point, called MQTT Broker, which takes care of dispatching all the messages between the senders and the rightful receivers.

When the clients publish messages to the broker, they include a topic into the message. For the broker, the topic acts as the routing information. Each client that wants to receive messages subscribes to a certain topic and the broker takes care of delivering all messages with the matching topic to the relevant client.

There is no requirement for the clients to know each other directly and the communication happens only over the topic. This pattern removes the dependencies of direct connectivity between the data producers and the data consumers and thus enables scalable solutions.

Apart from this, the protocol has the property of client side requiring small code footprint and less bandwidth, while the MQTT Broker can do heavy lifting of receiving messages from thousands of clients concurrently, filtering and routing each message to the right client who have subscribed to the topic.

This makes MQTT protocol ideal for resource constrained IoT devices which needs to be bandwidth-efficient and use little battery power.

# Relevant APIs

## BLE APIs

### 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.

### bt\_gatt\_create\_service\_16()

Creates a service declaration from a 16-bit UUID given as parameter and returns the pointer to the GATT service.

|  |
| --- |
| struct gatt\_service \* bt\_gatt\_create\_service\_16(uint16\_t uuid16) |

### 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) |

### 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) |

### 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 input:

1. adv\_fast\_period (ms): for this period, fast advertising is attempted every adv\_fast\_int interval. Once this period is over, 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).

This 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 is 125,000‬µs, which is, every 125ms, 8 times per second.

1. adv\_slow\_int In 625µs units: This sets the interval between two slow advertisements. Range: 0x0020 to 0x4000 (default: 1600).

This implies that 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 is 1,000,000µs, which is, every 1000ms, once per second.

1. adv\_tx\_power In dBm, range: -127 to 10, and 127 (127=no preference) (default: 127)
2. adv\_ch\_map Channel map used: bit0=ch37, bit1=ch38, bit2=ch39 (default: 0x7)

The API returns error code from bt\_gap\_error\_t.

### 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) |

Connection mode can be any one 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 inputs parameters to the API are as follows:

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.

### 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.

### 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.

### bt\_gatt\_add\_desc\_16()

Add a 16-bit UUID descriptor to a characteristic.

|  |
| --- |
| struct gatt\_desc \* bt\_gatt\_add\_desc\_16(struct gatt\_char \*c, uint16\_t uuid16, bt\_srv\_fcn\_t fcn, uint8\_t permission, uint8\_t property) |

### bt\_gap\_server\_link\_remove()

Add a 16-bit UUID descriptor to a characteristic.

|  |
| --- |
| struct gatt\_desc \* bt\_gatt\_add\_desc\_16(struct gatt\_char \*c, uint16\_t uuid16, bt\_srv\_fcn\_t fcn, uint8\_t  permission, uint8\_t property) |

## MQTT APIs

### MQTTNetworkInit()

Initializes MQTT network object with socket read, write, and disconnect functions.

|  |
| --- |
| void MQTTNetworkInit(MQTTNetwork\* n) |

### MQTTNetworkConnect()

Opens a socket and tries to connect the MQTT network object to the network endpoint.

|  |
| --- |
| int MQTTNetworkConnect(MQTTNetwork\* n, char\* addr, int port) |

### MQTTNetworkDisconnect()

Closes the socket and tries to connect the MQTT network object to the network endpoint.

|  |
| --- |
| void MQTTNetworkDisconnect(MQTTNetwork \*n) |

### MQTTClientInit()

Creates an MQTT client object.

|  |
| --- |
| void MQTTClientInit(MQTTClient\* client, MQTTNetwork\* network, unsigned int command\_timeout\_ms,unsigned char\* sendbuf, size\_t sendbuf\_size, unsigned char\* readbuf, size\_t readbuf\_size); |

### MQTTConnect()

Sends an MQTT connect packet down the network and wait for a Connack. The network object must be connected to the network endpoint before calling this.

|  |
| --- |
| int MQTTConnect(MQTTClient\* client, MQTTPacket\_connectData\* options); |

### MQTTDisconnect()

Sends an MQTT disconnect packet and closes the connection.

|  |
| --- |
| int MQTTDisconnect(MQTTClient\* client); |

### MQTTPublish()

Sends an MQTT publish packet and waits for all acks to complete.

|  |
| --- |
| int MQTTPublish(MQTTClient\* client, const char \*topic, MQTTMessage \*message); |

### MQTTSubscribe()

Send an MQTT subscribe packet and wait for suback before returning.

|  |
| --- |
| int MQTTSubscribe(MQTTClient\* client, const char\* topicFilter, enum QoS qos, MQTTMessageHandler messageHandler); |

### MQTTUnsubscribe()

Send an MQTT unsubscribe packet and wait for unsuback before returning.

|  |
| --- |
| int MQTTUnsubscribe(MQTTClient\* client, const char\* topicFilter); |

### MQTTYield()

MQTT goes to background for the time (ms) to yield for.

|  |
| --- |
| int MQTTYield(MQTTClient\* client, int time); |

# Application Flow

In this application, Talaria TWO is programmed to become a GATT server with a characteristic which can be read and written by a connected BLE client. The written text message is published to the MQTT Broker running as MQTT instance. A custom BLE service is also created to send the indications from Talaria TWO to the connected BLE client.

Following are the steps to achieve this:

1. Connect the device to a Wi-Fi network, whose SSID and passphrase are given as boot arguments while flashing the binary image.
2. Connect to the MQTT instance using URL, port, username, and password of the cloud which are also given as boot arguments.
3. Initialize GAP profile and create GATT services for reading the name of BLE device and writing some text to the BLE peripheral device.
4. Create a BLE custom service to send the indications to BLE client.
5. On receiving any texts onto the BLE device, publish it to the MQTT client.
6. The published messages can be seen on subscriber’s console window.
7. Upon receiving any messages for the topics subscribed, Talaria TWO will send the messages as indications to the connected BLE Client.

# Code Walkthrough

## Reading the Boot Argument

While programming the elf to Talaria TWO using boot.py, we use bootargs to pass below necessary parameters to the program.

1. SSID and Passphrase of Wi-Fi Network
2. URL, Port, Username and Password of the MQTT server

In main(), first these parameters are retrieved as shown below.

main.c

|  |
| --- |
| int main()  {  print\_ver("Ble Wifi Bridge Demo App", 1, 3);  struct network\_profile \*profile;  int rval;  int ret;  const char \*ssid = os\_get\_boot\_arg\_str("ssid");  const char \*passphrase = os\_get\_boot\_arg\_str("passphrase")?:NULL;  const char \*np\_conf\_path = os\_get\_boot\_arg\_str("np\_conf\_path")?: NULL;  cloud\_url = os\_get\_boot\_arg\_str("cloud\_url") ;  cloud\_port = os\_get\_boot\_arg\_int("cloud\_port", 1883) ;  cloud\_usr\_name = os\_get\_boot\_arg\_str("cloud\_usr\_name") ;  cloud\_usr\_psw = os\_get\_boot\_arg\_str("cloud\_usr\_psw") ?: ""; |

## Connecting to a Wi-Fi Network

To connect to a Wi-Fi network, wcm\_create()API from the Wi-Fi Connection Manager are used.

main.c

Initially, the Wi-Fi network interface is created using wcm\_create().

|  |
| --- |
| h = wcm\_create(NULL); |

wifi\_connect\_to\_network()API, from components library, connects to the Wi-Fi network using the AP credentials provided.

|  |
| --- |
| rval = wifi\_connect\_to\_network(&h, WCM\_CONN\_WAIT\_INFINITE, &wcm\_connect\_success);  if(rval < 0) {  os\_printf("\nError: Unable to connect to network\n");  return 0;  } |

Once the Wi-Fi connection is successful, the application begins the BLE Wi-Fi bridge services.

|  |
| --- |
| start\_ble\_wifi\_bridge\_services(); |

The on\_new\_message\_via\_ble() function publishes the message received from the BLE device.

|  |
| --- |
| int on\_new\_message\_via\_ble(char \*message, int len)  {  vTaskDelay(100); /\* added wait for 100 m sec before publishing\*/  return (bmw\_publish\_message(message, len));  } |

my\_app\_init() function creates a thread for BLE, subscribe and publish operations.

## Initializing MQTT Client

Once connection to a Wi-Fi network is established, connection to the MQTT broker can be initiated

In main.c, start\_mqtt() is called and the login parameters of the MQTT broker are passed.

In start\_mqtt(), the handle to MQTT network is created by allocating an MQTT network object mqtt\_network of type MQTTNetwork.

Then this network object is initialized using MQTTNetworkInit()and a connection is established to network endpoint by passing the initialized handle mqtt\_network, cloud\_url and cloud\_port to MQTTNetworkConnect().

|  |
| --- |
| static void start\_ble\_wifi\_bridge\_services(void)  {  if(start\_mqtt((char \*)cloud\_url, cloud\_port, (char \*)cloud\_usr\_name, (char \*)cloud\_usr\_psw) == 0 ){  os\_printf("starting subscriber\_publisher\_thread\n");  start\_ble();  bmw\_subscribe\_message();  }else{  if(restart\_mqtt\_connection() == 0){  start\_ble();  bmw\_subscribe\_message();  }else{  os\_printf("Check if the MQTT broker is active\r\n");  }  }  } |

MQTT Client is initialized in the same code.

Object of type MQTTClient is allocated first and held by handle mqtt\_client.

Then MQTTClientInit() is called with the MQTT network object mqtt\_network, pointers to read and send buffers and the MQTTClient handle mqtt\_client.

mqtt.c

|  |
| --- |
| static MQTTNetwork \*mqtt\_network;  static MQTTClient \*mqtt\_client;  . . .  . . .  int start\_mqtt(char \*cloud\_url, int cloud\_port, char \*cloud\_usr\_name, char \*cloud\_usr\_psw)  {  int rc;  /\*initializing MQTT\*/  mqtt\_network= osal\_zalloc(sizeof(MQTTNetwork));  MQTTNetworkInit(mqtt\_network);  . . .  . . .  rc = MQTTNetworkConnect(mqtt\_network, cloud\_url, cloud\_port);  if(rc != 0)  {  os\_printf(“\nMQTTNetworkConnect failed ret:%d (%s)\n”, rc, strerror(rc));  return (void\*)-1;  }  . . .  . . .  return rc;  } |

Then, a connect request packet is made using client\_data.client\_id as T2\_<mac\_id>, cloud\_usr\_name and cloud\_usr\_psw.

And finally, a connection request to MQTT broker is made with the packet made in above step, using MQTTConnect(). As shown in the following section:

mqtt.c

|  |
| --- |
| int start\_mqtt(char \*cloud\_url, int cloud\_port, char \*cloud\_usr\_name, char \*cloud\_usr\_psw)  {  int rc;  . . .  . . .  mqtt\_client = osal\_zalloc(sizeof(MQTTClient));  /\* Opens a socket and tries to connect the MQTT network  object to the network endpoint. \*/  rc = MQTTNetworkConnect(mqtt\_network, cloud\_url, cloud\_port);  . . .  . . .  generate\_mqtt\_topics(&client\_data);  /\* Creates an MQTT client object. \*/  MQTTClientInit(mqtt\_client, mqtt\_network, TIMEOUT\_MS,  sendbuf, sizeof(sendbuf), readbuf, sizeof(readbuf));  MQTTPacket\_connectData data = MQTTPacket\_connectData\_initializer;  data.willFlag = 0;  data.MQTTVersion = 3;  data.username.cstring = cloud\_usr\_name;  data.password.cstring = cloud\_usr\_psw;  data.clientID.cstring = client\_data.client\_id;  os\_printf("\*\*\*MQTT Client id is %s\r\n",data.clientID.cstring);  data.keepAliveInterval = MQTT\_KEEP\_ALIVE\_INTERVAL; //default was 100  data.cleansession = 0;  os\_printf("Connecting to %s:%d\n", cloud\_url, cloud\_port);  /\* Sends an MQTT connect packet down the network and wait for a Connect.  The network object must be connected to the network endpoint  before calling this. \*/  mqtt\_client->ping\_outstanding = 0;  rc = MQTTConnect(mqtt\_client, &data);  os\_printf("Connected to %s:%d ret:%d\n", cloud\_url, cloud\_port, rc);  return rc;  } |

fetch\_t2\_macid()function fetches the mac ID of the Talaria TWO device and stores it in mac\_id[index] buffer.

|  |
| --- |
| const uint8\_t \*mac\_addr = wcm\_get\_hwaddr(h);  os\_printf("mac id:");  for(int index =0;index<6;index++){  mac\_id[index] = \*(mac\_addr+index);  os\_printf("%x",mac\_id[index]); |

generate\_mqtt\_topics() function generates the MQTT publish and subscribe topics by using the mac ID of Talaria TWO and prints these topics onto the console. The other MQTT client (PC in this example) can subscribe/publish to these topics.

|  |
| --- |
| uint8\_t t2\_mac\_id[LEN\_OF\_MAC\_ID];  uint8\_t len\_mqtt\_sub\_topic,len\_mqtt\_pub\_topic;  char temp\_buf[10];  int index = 0;  fetch\_t2\_macid(t2\_mac\_id);  for (int i=0; i < LEN\_OF\_MAC\_ID; i++){  index += snprintf(&temp\_buf[index], 128-index, "%x", t2\_mac\_id[i]);  }  sprintf(mqtt\_client\_data->client\_id,"T2\_%s",temp\_buf);  len\_mqtt\_sub\_topic = strlen(mqtt\_client\_data->client\_id)+strlen(topic\_subscribe)+1;  snprintf(mqtt\_client\_data->subscribe\_topic,len\_mqtt\_sub\_topic,"%s%s",mqtt\_client\_data->client\_id,topic\_publish);  len\_mqtt\_pub\_topic = strlen(mqtt\_client\_data->client\_id)+strlen(topic\_publish)+1;  snprintf(mqtt\_client\_data->publish\_topic,len\_mqtt\_pub\_topic,"%s%s",mqtt\_client\_data->client\_id,topic\_subscribe);  os\_printf("\r\n------------------------------------------------------\r\n");  os\_printf("MQTT Client id : %s\r\n",mqtt\_client\_data->client\_id);  os\_printf("MQTT publish topic: %s\r\n", mqtt\_client\_data->publish\_topic);  os\_printf("MQTT subscribe topic: %s\r\n", mqtt\_client\_data->subscribe\_topic);  os\_printf("--------------------------------------------------------\r\n"); |

## Publishing Data to the MQTT Instance

Function bmw\_publish\_message()takes pointer and length of a message as input and publishes it to the remote MQTT Broker running as MQTT instance.

Message is published using the MQTTPublish() function under the topic T2\_<mac id of T2>/subscribe.

mqtt.c

|  |
| --- |
| int bmw\_publish\_message(char \*pmessage, int len)  {  int rc = 0;  MQTTMessage \*publish = osal\_zalloc(sizeof(MQTTMessage));  publish->payload = pmessage;  publish->payloadlen =len;  memcpy(device\_data\_recieved, pmessage, len);  device\_data\_recieved[len]='\0';  os\_printf("\n\n%u:from BLE Client, Message Recieved[%s]",os\_systime(), device\_data\_recieved);  if(mqtt\_connection\_status\_check() == 1){  os\_printf("\nMQTT connection is Active");  restarting\_session = false;  }  else{  restart\_mqtt\_connection();  }  rc = MQTTPublish(mqtt\_client, client\_data.publish\_topic, publish); if(rc != 0)  {  os\_printf("\nMQTTPublish failed. Ret= %d", rc);  }  else  {  os\_printf("\n%u:Message published successfully [%s]",os\_systime(), pmessage);  }  osal\_free(publish);  return 0;  } |

## Subscribing to MQTT Topic

Function bmw\_subscribe\_message()subscribes to a topic and registers the call back function MQTTSubscribeCallback (MessageData\* Msg). The call back gets invoked when there is a message published by a client to the same topic.

|  |
| --- |
| void MQTTSubscribeCallback(MessageData\* Msg)  {  os\_printf("\nMQTTSubscribe Call back:%s \n",(char\*)Msg->message->payload);  send\_ble\_indications((uint8\_t\*)Msg->message->payload,Msg->message->payloadlen);  memset((char\*)Msg->message->payload,0,Msg->message->payloadlen);  }  int bmw\_subscribe\_message(void)  {  MQTTYield(mqtt\_client, 1000);  MQTTMessageHandler messageHandler = &MQTTSubscribeCallback;  MQTTSubscribe(mqtt\_client, client\_data.subscribe\_topic, QOS1, messageHandler);  return 0;  } |

bmw\_unsubscribe\_message()function unsubscribes to the topic which is already subscribed to.

|  |
| --- |
| int bmw\_unsubscribe\_message(void)  {  MQTTUnsubscribe(mqtt\_client, client\_data.subscribe\_topic);  return 0;  } |

bmw\_Yield()function yields to check if any messages are to be published/subscribed.

|  |
| --- |
| void bmw\_Yield(void)  {  MQTTYield(mqtt\_client, 500);  } |

## Running BLE GATT Server

Once connection with the MQTT network is established, BLE GAP is initialized and a GATT service with a write only characteristic is started.

The message written by any connected BLE client to this characteristic is published to MQTT broker. A smartphone application is used in this example to write a message to this characteristic.

Initially, the Bluetooth GAP service is initialized using bt\_gap\_init() and then a custom Bluetooth GATT service is created.

The bt\_gatt\_create\_service\_16() function creates a custom GATT service with a 16-bit UUID.

|  |
| --- |
| bt\_gap\_error\_t custom\_ind\_server\_create(void)  {  srv16 = bt\_gatt\_create\_service\_16(CUSTOM\_IND\_SERVICE\_UUID);  chr\_i = bt\_gatt\_add\_char\_16(srv16, CUSTOM\_IND\_SERVICE\_CHARACTERISTIC\_UUID, NULL, 0, GATT\_CHAR\_PROP\_I);  bt\_gatt\_add\_desc\_16(chr\_i, UUID\_GATT\_CD\_CLIENT\_CONFIGURATION, indication\_cccd, GATT\_PERM\_RW, GATT\_CHAR\_PROP\_RW);  bt\_gatt\_add\_service(srv16);  return GAP\_ERROR\_SUCCESS;  } |

bt\_gatt\_add\_char\_16() is used to add a characteristic with a 16-bit UUID to that service. Callback function pointer is provided as parameter to this function which will be called when the characteristic is accessed. Properties and permissions for the characteristic are also specified.

Following are two such characteristics that are added:

1. UUID\_GATT\_CT\_DEVICE\_NAME and the callback associated when accessing this characteristic is ‘device\_name\_read()’.
2. The other one is write only and the callback associated when accessing this characteristic is ‘data\_receive()’. In example UUID\_GATT\_CT\_DEVICE\_NAME+1 is used which is UUID\_GATT\_CT\_APPEARANCE.

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

ble.c

|  |
| --- |
| // Initialize GAP and GATT  int start\_ble()  {  bt\_gap\_init();  create\_my\_bt\_service();  return 0;  }  // Create GATT service  bt\_gap\_error\_t create\_my\_bt\_service()  {  bt\_gap\_cfg\_adv\_t bt\_adv\_handle;  if (mqtts)  return GAP\_ERROR\_SUCCESS;  mqtts = osal\_zalloc(sizeof(mqtts\_t));  mqtts->srv = bt\_gatt\_create\_service\_16(UUID\_GATT\_S\_GENERIC\_ACCESS);  bt\_gatt\_add\_char\_16(mqtts->srv, UUID\_GATT\_CT\_DEVICE\_NAME, device\_name\_read,  GATT\_PERM\_READ, GATT\_CHAR\_PROP\_R); /\* \_REA\*/  bt\_gatt\_add\_char\_16(mqtts->srv, UUID\_GATT\_CT\_DEVICE\_NAME+1, data\_receive,  GATT\_PERM\_WRITE, GATT\_CHAR\_PROP\_W);  . . .  . . .  } |

Here bt\_gap\_cfg\_adv() sets parameters for advertisement. The parameters passed for configuring the advertisement are as follows:

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

This means the fast advertising will be attempted for nearly 10 seconds (10.24s) when advertisement is enabled. After this 10.24s period, the slow advertisement will be attempted.

1. adv\_slow\_period is set to 0, this means 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 means (160\*625µs) = 100,000µs = every 100ms is the interval at which fast advertisement will be attempted.
3. adv\_slow\_int is set to 480, which means (480\*625µs) = 300,000‬µs= every 300ms will be the interval of slow advertising.
4. bt\_gap\_cfg\_smp() is used to set security parameters. Here it is passed as 0, so no security parameter is configured.
5. bt\_gap\_connectable\_mode() makes the device connectable and will enable advertisement.

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

ble.c

|  |
| --- |
| // Create GATT service  bt\_gap\_error\_t create\_my\_bt\_service()  {  . . .  custom\_ind\_server\_create();  bt\_gatt\_add\_service(mqtts->srv);  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);  /\*return gap connectable mode\*/  return bt\_gap\_connectable\_mode(GAP\_CONNECTABLE\_MODE\_UNDIRECT,  bt\_hci\_addr\_type\_random, 0, address\_zero, &gap\_ops);  } |

## 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:

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

The code sample shows 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 using bt\_gap\_server\_link\_remove() when the client disconnects,

|  |
| --- |
| bt\_gap\_server\_link\_remove(mqtts->gatt); |

## Characteristic Access Callback

While the client is connected to the server, it can read or write the custom characteristic based on the characteristic’s properties. This results in the callback function associated with the characteristic being called; in this case, device\_name\_read() and data\_receive().

When the read only characteristic UUID\_GATT\_CT\_DEVICE\_NAME is accessed, the callback associated when accessing this characteristic device\_name\_read() is called.

It passes inno\_mqtt as device name to the client reading this characteristic.

ble.c

|  |
| --- |
| static bt\_att\_error\_t device\_name\_read(uint8\_t bearer, bt\_uuid\_t \*uuid, bt\_gatt\_fcn\_t rw, uint8\_t \*length, uint16\_t offset, uint8\_t \*data)  {  char device\_index\_str[15];  snprintf(device\_index\_str,sizeof(device\_index\_str),"inno\_mqtt");  uint8\_t len = strlen(device\_index\_str) - offset;  if (offset >= len)  return BT\_ATT\_ERROR\_INVALID\_OFFSET;  if (\*length > len)  \*length = len;  memcpy(data, device\_index\_str, \*length);  os\_printf ("\n BLE Device Name Read callback -- [%s]\n", device\_index\_str);  return BT\_ATT\_ERROR\_SUCCESS;  } |

When the write only characteristic UUID\_GATT\_CT\_DEVICE\_APPEARANCE is accessed, the callback associated when accessing this characteristic data\_receive() is called. BLE GATT server receives the text messages from BLE Client.

ble.c

|  |
| --- |
| static bt\_att\_error\_t  data\_receive(uint8\_t bearer, bt\_uuid\_t \*uuid, bt\_gatt\_fcn\_t rw, uint8\_t \*length, uint16\_t offset, uint8\_t \*data)  {  on\_new\_message\_via\_ble((char \*)data, \*length);  return BT\_ATT\_ERROR\_SUCCESS;  } |

Function on\_new\_message\_via\_ble()calls bmw\_publish\_message() in main.c, described in section 7.4, and the message written is published to broker under the topic innophase\_<T2 mac id>/publisher.

# Running the Application using Mosquitto Project’s Test Server

Eclipse Mosquitto is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 5.0, 3.1.1 and 3.1.

The Mosquitto project allows to test the MQTT based applications to test using its test server. Users can use a custom server or any of the following tested public MQTT brokers:

1. **mqtt.eclipseprojects.io**
   1. 1883 : MQTT over unencrypted TCP
   2. 8883 : MQTT over encrypted TCP
   3. 80 : MQTT over unencrypted Websocket (note: URL must be */mqtt* )
   4. 443: MQTT over encrypted WebSockets (note: URL must be */mqtt* )
2. **mqtt-dashboard.com**
   1. TCP Port: 1883
   2. TLS TCP Port: 8883
   3. Websocket Port: 8000
   4. TLS Websocket Port: 8884
3. test.mosquitto.org
   1. 1883: MQTT, unencrypted, unauthenticated
   2. 1884: MQTT, unencrypted, authenticated
   3. 8883: MQTT, encrypted, unauthenticated
   4. 8884: MQTT, encrypted, client certificate required
   5. 8080: MQTT over WebSockets, unencrypted, unauthenticated
   6. 8081: MQTT over WebSockets, encrypted, unauthenticated
   7. 8091: MQTT over WebSockets, unencrypted, authenticated

**Note**: test.mosquitto.org is used in this document for illustration purposes only.

The following steps describes the procedure to test the ble\_wifi\_bridge application using Mosquitto project’s test server.

## Installing and Running the Mosquitto MQTT Tool

1. Download [mosquitto-2.0.11-install-windows-x64.exe](file:///C:\C:\Users\91963\Downloads\mosquitto-2.0.11-install-windows-x64.exe) from <https://mosquitto.org/download/> and install the same
2. Open a command prompt window on the PC and subscribe to a topic by issuing the following command:

|  |
| --- |
| mosquitto\_sub.exe -h test.mosquitto.org -p 1883 -u <user name> -P <Password> -t T2\_<mac\_id>/publisher |

Ensure that the note in section 7.4 is followed and the binary is generated.

In the example, username and password used is innophase. The pub/sub topics are computed by the application for Talaria TWO module. For this example they are: T2\_e0693a015b2/publisher and T2\_e0693a015b2/subscribe. Ensure to check the console logs on the Download Tool for the publish/subscribe topics based on the unique MQTT client id generated for the Talaria TWO device.

Figure 1 shows the command prompt window:



Figure 1: Command prompt window

## Programming the Talaria TWO Module

Program wifi\_ble\_mqtt.elf*(freertos\_sdk\_x.y\examples\ble\_wifi\_bridge\bin)* usingthe Download tool:

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 ELF by clicking on Select ELF File.
   3. AP Options: Provide the SSID and Passphrase under AP Options to connect to an Access Point.
   4. Boot arguments: Pass the following boot arguments:

|  |
| --- |
| cloud\_url=test.mosquitto.org,cloud\_port=1883,cloud\_usr\_name=<user name >,cloud\_usr\_psw=<password> |

* 1. Programming: Prog RAM or Prog Flash as per requirement.

The console should display a return value of 0 indicating that the Talaria TWO is able to connect to test.mosquitto.org server.

Console output:

|  |
| --- |
| Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PWWWWWWWAE  Build $Id: git-e52d93e $  Flash detected. flash.hw.uuid: 39483937-3207-0080-0055-ffffffffffff  Bootargs: ssid=Rczz\_2.4G passphrase=rc@9980044013 cloud\_usr\_name=innophase cloud\_usr\_psw=innophase cloud\_url=mqtt.eclipseprojects.io  SDK Ver: FREERTOS\_SDK\_1.0  Ble Wifi Bridge Demo App  addr e0:69:3a:00:08:38  network profile created for ssid: Rczz\_2.4G  Connecting to added network : Rczz\_2.4G  [0.920,765] CONNECT:70:4f:57:4a:fc:85 Channel:11 rssi:-57 dBm  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_LINK\_UP  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_ADDRESS  [1.648,724] MYIP 192.168.0.116  [1.648,888] IPv6 [fe80::e269:3aff:fe00:838]-link  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_CONNECTED  Connected to added network : Rczz\_2.4G  /home/osboxes/Work/Freertos/dev/freertos\_embedded\_apps/components/mqtt/platform/mqtt\_nw\_tcp.c:MQTTNetworkConnectmac id:e0693a0838  ------------------------------------------------------  MQTT Client id : T2\_e0693a0838  MQTT publish topic: T2\_e0693a0838/subscribe  MQTT subscribe topic: T2\_e0693a0838/publisher  --------------------------------------------------------  \*\*\*MQTT Client id is T2\_e0693a0838  Connecting to mqtt.eclipseprojects.io:1883  \_mqtt\_cycle : packet\_type = 2Connected to mqtt.eclipseprojects.io:1883 ret:0  starting subscriber\_publisher\_thread  BLE Device Name Read callback -- [inno\_mqtt]  \_mqtt\_cycle : packet\_type = 9[37.233,796] BT connect[0]: ia:72:1e:ff:02:d9:73 aa:00:00:00:00:00:00 phy2:0/0 phyC:00  connected!  BLE Device Name Read callback -- [inno\_mqtt] |

## Publishing a Topic and Sending Data from BLE to Wi-Fi

1. Open the BLE application on the mobile phone and click on the Scan button at the top right corner. Look for the device inno\_mqtt.

**Note**: In the example, a mobile application called nRF Connect for Mobile, developed by Nordic Semiconductor ASA is used.

1. Click on the CONNECT button adjacent to the device name inno\_mqtt to connect to the device. This will show the GATT services and custom service (shown as unknown service in nRF Connect app) in a new window.

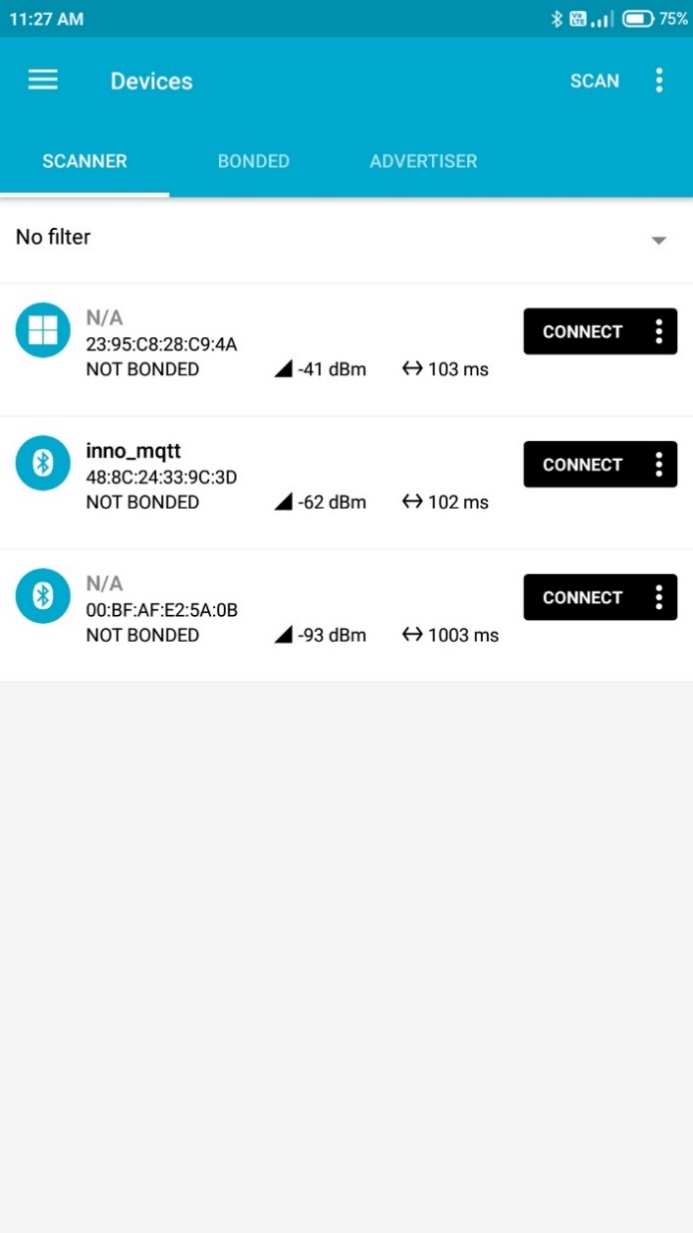


Figure 2: Publishing a topic - Connecting to inno\_mqtt

1. Click on GENRIC ACCESS->WRITE as shown in Figure 3.

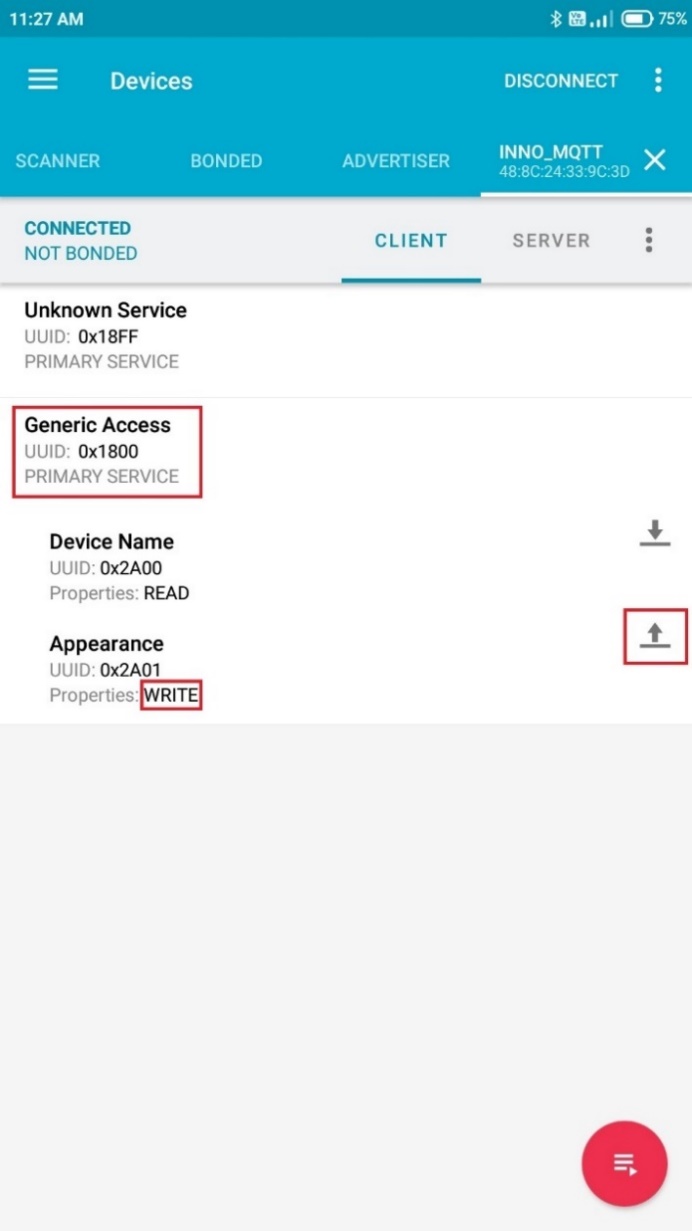


Figure 3: Publishing a topic - Generic Access - W

1. Sending a message using the nRF Connect app: Select the message type from the drop-down, write a message and click SEND.

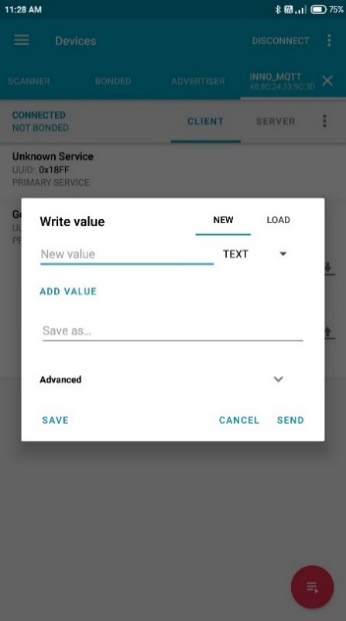
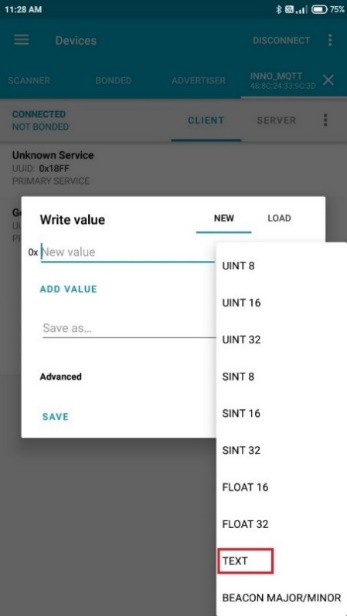


Figure 4: Selecting message type

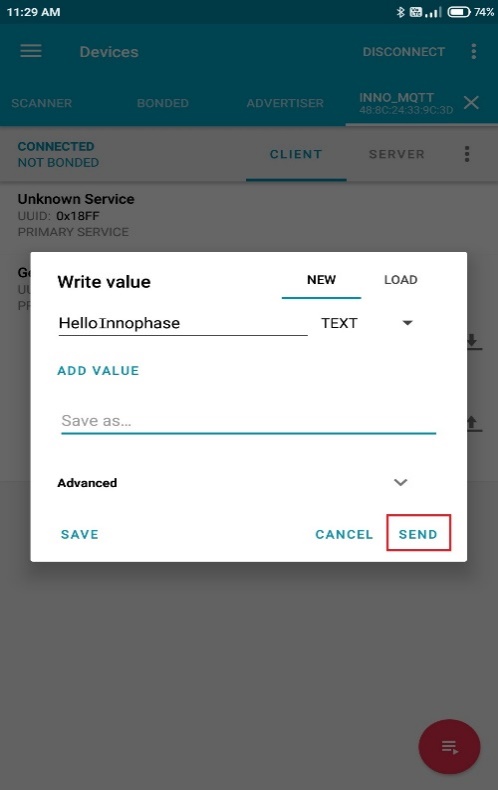


Figure 5: Sending a message

1. If message size is more than 30bytes, it is required to change the MTU size in nRF Connect Mobile application.

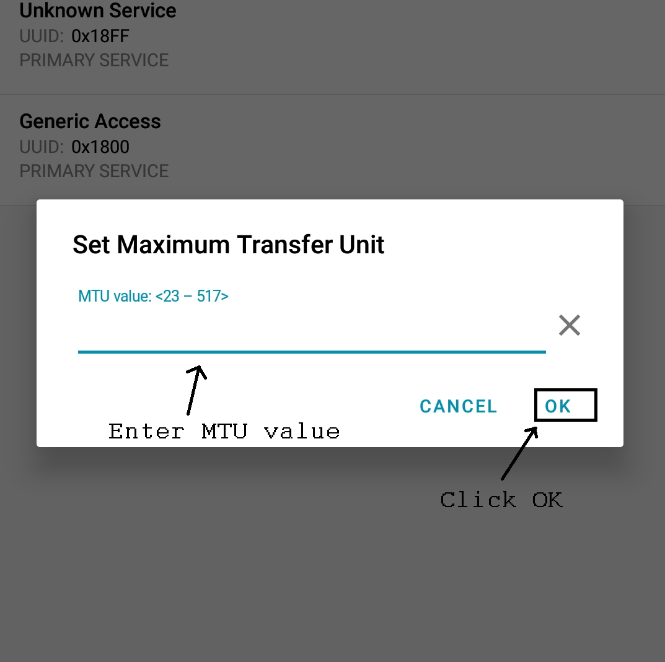
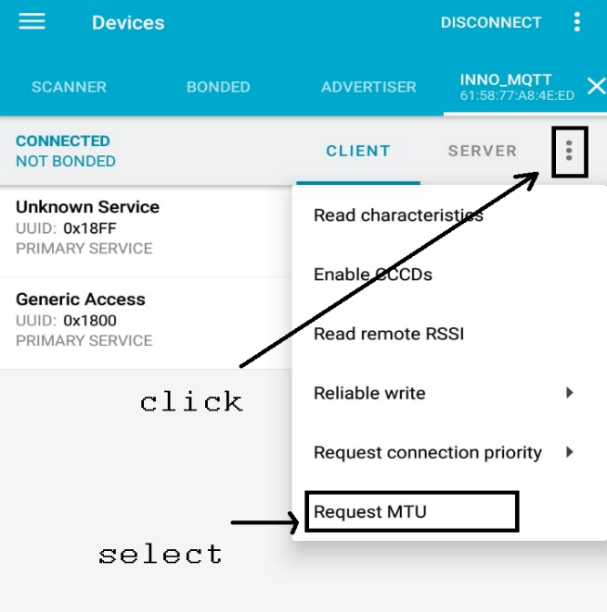


Figure 6: Changing MTU size

1. The message will be published by Talaria TWO and the same is observed on the Download Tool’s console window as well indicating that the message was successfully published.

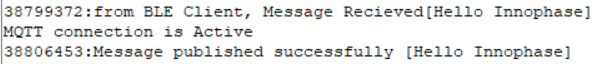


Figure 7: Publishing a topic - Output

1. The published message by Talaria TWO can be seen on the subscriber’s command prompt window opened during step 2 of section 8.1.



Figure 8: Publishing a topic - Command prompt output

## Subscribing to a Topic and Sending Data from Wi-Fi to BLE

1. Open the BLE application on the mobile phone and click on the Scan button at the top right corner and look for the device inno\_mqtt.

**Note**: In the example, a mobile application called nRF Connect for Mobile, developed by Nordic Semiconductor ASA is used.

1. Click on the CONNECT button adjacent to the device name inno\_mqtt to connect to the device. This will show the GATT services and custom service (shown as unknown service in nRF connect app) in a new window.

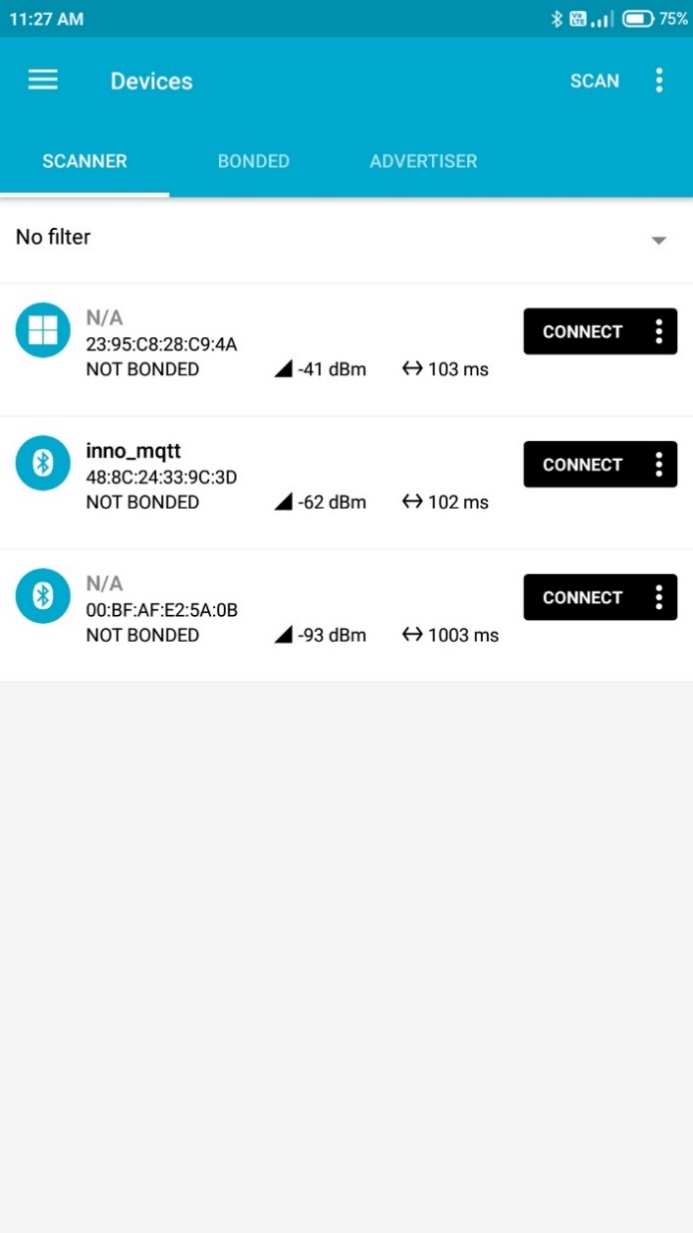


Figure 9: Subscribing to a topic - Connecting to inno\_mqtt

1. Click on Unknown Service (Download symbol) as shown in Figure 10.

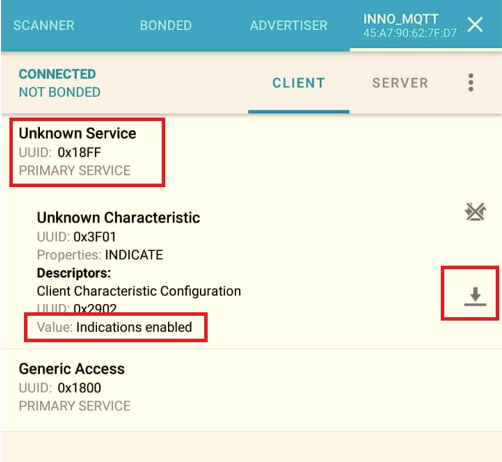


Figure 10: Subscribing to a topic - Generic Access - I

1. Publish a message from any MQTT client to the topic T2\_<mac\_id>/publisher. The publish topic computed by the application for Talaria TWO module used in this application is T2\_e0693a015b2/publisher.

Ensure to check the console logs on the Download Tool for the publish topic based on the unique MQTT client id generated for the Talaria TWO device.



Figure 11: Publishing a message from MQTT client

1. Since Talaria TWO device has subscribed to the topic T2\_e0693a015b2/subscribe, the subscribe topic computed by the application for one of the Talaria TWO module is T2\_e0693a015b2/subscribe. Ensure to check the console logs on the Download Tool for the subscribe topic based on the unique MQTT client ID generated for the Talaria TWO device. The same can be observed on Talaria TWO’s console:



Figure 12: Subscribing to a topic - output

1. Talaria TWO sends the received message over BLE to the BLE client. This message will be displayed on the nRF connect app.

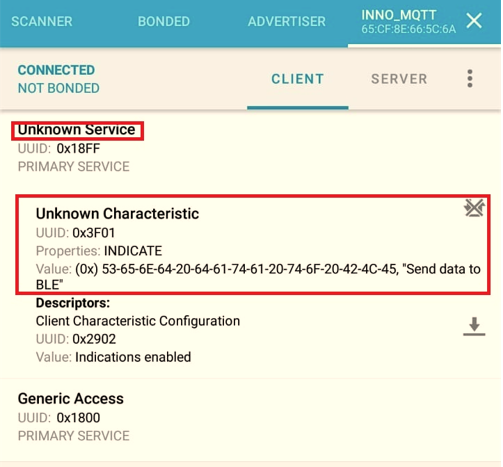


Figure 13: Message sent to BLE Client over BLE