MT793X IoT SDK for Bluetooth Mesh Developer's Guide

Version: 0.2

Release date: 2021-08-13

Version History

Version	Date	Description
0.1	2021-03-19	Initial draft
0.2	2021-08-13	Adjust document template and fix some typo

Table of Contents

List of Figures

List of Tables

1 Overview

This document introduces the Bluetooth (BT) mesh networking features that are supported by the MT793x IoT SDK and shows how to customize the application code for mesh functions.

Note: In MT793X SDK, MediaTek can only support the role of Mesh Provisionee now on SDK1.0.
For more detailed feature list please contact MediaTek's representative.

1.1 Basic Concept

Network Topology and Subnet

Bluetooth mesh is a broadcast-based network protocol. Every device is able to directly send and receive messages to and from other devices within radio range. To extend the range of communication, a device can relay messages to a different device outside the radio range and receive messages that are relayed. Therefore, all devices that have at least one other device within radio range can construct a mesh network, even when a device is physically relocated.

A mesh network shares mutually-known network keys to secure and authenticate messages at the network layer. Application keys secure and authenticate messages at the access layer.

A subnet of a mesh network is formed when only a small group of devices share one of the network keys. A device can be a part of one or more subnets if it knows more than one of the network keys.

Mesh Transport

Bluetooth mesh sends messages through Bluetooth Low Energy advertisement, and receives messages by scanning the advertisements. Mesh packets advertise with specific AD types defined by the Bluetooth SIG, so that those scanned packets can be recognized as Mesh transport.

In a managed-flood-based mesh network, there are two primary methods for restricting the unlimited relay of messages: One is network message cache; the other is Time to live (TTL) method.

The network cache is designed to prevent a device from relaying the same message every time it scans the message.

The Time to Live (TTL) value is a field included in every mesh message to specify the number of times that a message can be relayed.

Low Power Node and Friend Node

As previously mentioned, Bluetooth mesh is power-consuming. Hence, the feature of low power nodes and friend nodes are designed to help battery-powered devices conserve energy.

A low power node can make friends with a friend node. When the friendship is established, the friend node is

responsible for receiving and sending mesh packets on behalf of the low power node when the low power node goes into hibernation.

Provisioning

A device is known as an "unprovisioned device" before it is made a node of a mesh network. Devices must be added by a Provisioner (i.e. the manager of a mesh network) to join a Mesh network. During the provisioning process, devices are given a network key. The node can then communicate with other nodes that have the same network key.

Elements and Models

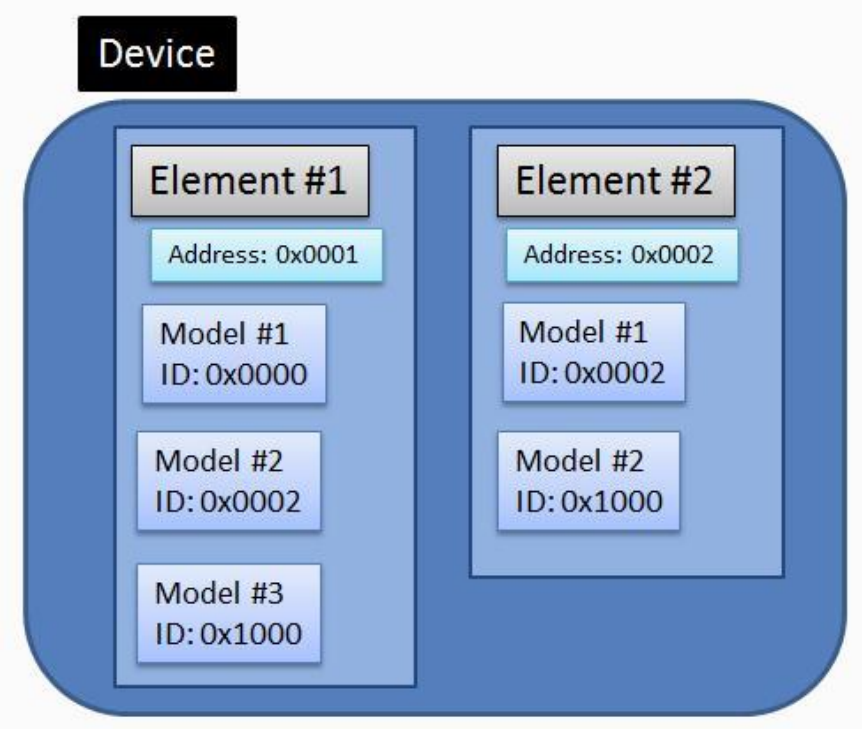

Figure 1. Example of a mesh device composed of two elements

An element is an addressable unit within a node. Each node has at least one element, the primary element, which uses the first unicast address assigned to the node during provisioning. A model is a unit that defines the basic functionality, and is included in an element. An element can have multiple models with different model IDs. Figure 1 shows an example of a mesh device composed of two elements. Each element consists of several different models.

Server, Client, and Control Models

Communications between two models of the nodes are based on client-server architecture. A typical scenario is

that a device with a specific client model sends a set/get message to another device with a corresponding server model responding with a status message. Models are categorized as foundation models, mesh models, and vendor-defined models. Foundation models are mandatory models. Mesh models are dependent on the abilities of the device.

1.2 Bluetooth Mesh Stack Architecture

The Bluetooth mesh stack architecture is shown in Figure 2. Bluetooth mesh stack architecture

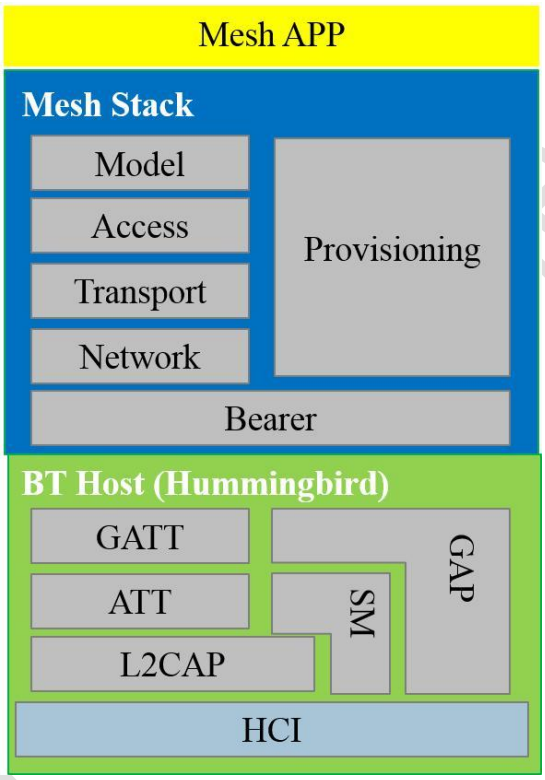

Figure 2. Bluetooth mesh stack architecture

2 Bluetooth Mesh Support

The MT793x IoT SDK is designed to support SIG Mesh features such as the following:

- Provision via ADV (or GATT)
- Relay Node
- Proxy Node
- Friend Node
- Mesh Security
- Mesh Beacons

The IoT SDK allows you to write applications for both the provisioner and provisionee roles, with various ready-to-use models such as Light server models and Scene server models as defined in the SIG Mesh Model specification, as shown in Table 1. Supported models in IoT SDK. Furthermore, the SDK provides APIs from which you can create customized models based on the features of specific devices.

Table 1. Supported models in IoT SDK

Model Group	Model Name	SIG Model ID	Supported in SDK
Generic	Generic OnOff Server	0x1000	o
	Generic OnOff Client	0x1001	o
	Generic Level Server	0x1002	o
	Generic Level Client	0x1003	o
	Generic Default Transition Time Server	0x1004	o
	Generic Default Transition Time Client	0x1005	o
	Generic Power OnOff Server	0x1006	o
	Generic Power OnOff Setup Server	0x1007	o
	Generic Power OnOff Client	0x1008	o
	Generic Power Level Server	0x1009	o

	Generic Power Level Setup Server	0x100A	o
	Generic Power Level Client	0x100B	o

Model Group	Model Name	SIG Model ID	Supported in SDK
	Generic Battery Server	0x100C	o
	Generic Battery Client	0x100D	o
	Generic Location Server	0x100E	o
	Generic Location Setup Server	0x100F	o
	Generic Location Client	0x1010	o
	Generic Admin Property Server	0x1011	o
	Generic Manufacturer Property Server	0x1012	o
	Generic User Property Server	0x1013	o
	Generic Client Property Server	0x1014	o
	Generic Property Client	0x1015	o
Sensors	Sensor Server	0x1100	o
	Sensor Setup Server	0x1101	o
	Sensor Client	0x1102	x
Time and Scenes	Time Server	0x1200	o
	Time Setup Server	0x1201	o
	Time Client	0x1202	x
	Scene Server	0x1203	o

	Scene Setup Server	0x1204	o
	Scene Client	0x1205	x
	Scheduler Server	0x1206	o
	Scheduler Setup Server	0x1207	o
Lighting	Light Lightness Server	0x1300	o
	Light Lightness Setup Server	0x1301	o
	Light Lightness Client	0x1302	o
	Light CTL Server	0x1303	o
	Light CTL Setup Server	0x1304	o

Model Group	Model Name	SIG Model ID	Supported in SDK
	Light CTL Client	0x1305	o
	Light CTL Temperature Server	0x1306	o
	Light HSL Server	0x1307	o
	Light HSL Setup Server	0x1308	o
	Light HSL Client	0x1309	o
	Light HSL Hue Server	0x130A	o
	Light HSL Saturation Server	0x130B	o
	Light xyL Server	0x130C	o
	Light xyL Setup Server	0x130D	o
	Light xyL Client	0x130E	x
	Light LC Server	0x130F	x
	Light LC Setup Server	0x1310	x
	Light LC Client	0x1311	x

2.1 IoT SDK Library

The SDK provides a library file interface to the Bluetooth mesh with C source and header files related to the platform, as shown in Table 2. MediaTek IoT SDK library support for Bluetooth mesh.

Table 2. MediaTek IoT SDK library support for Bluetooth mesh

Module	File Name	Location	Function
mesh stack	`libbt_mesh_stack.a`	`prebuilt/middleware/MTK/mesh/lib/`	Bluetooth mesh stack and SIG models library
mt7697 bearer	`libbt_mesh_bearer_7697.a`	`prebuilt/middleware/MTK/mesh/lib/`	Bluetooth mesh bearer library for the MT7697 chip
aw7933 bearer	`libbt_mesh_bearer_7933.a`	`prebuilt/middleware/MTK/mesh/lib/`	Bluetooth mesh bearer library for the MT793x chip

3 Implementation of Bluetooth Mesh Application

This chapter provides information about how to create your applications to build a proper mesh network. In general, two types of applications are required: one is the provisioner and the other is the provisionee.

3.1 Handling Message of Mesh Application

The incoming messages are received in the handler array used in *bt_mesh_model_add_model()*.

Use *bt_mesh_access_model_reply_ex()* to reply to that message. It uses the content of the received messages as a parameter. Use *bt_mesh_send_packet()* to independently send a message. The content and format of the payload in the parameter are defined according to the Bluetooth SIG specification.

3.2 Writing an Application of the Provisioner

3.2.1 Initialization

- **Initialize BT and create models of the device** – Initialize Bluetooth before creating anything for the mesh networking. For BT initialization, please refer to another develop guide <MT793X IoT SDK Bluetooth Developer's Guide.pdf>.
- **Initialize mesh** – Models cannot be added until the BT module is successfully initialized (i.e. when the event of BT_POWER_ON_CNF has been passed to callback that is registered before).

The following example shows when the mesh-related initialization can start.

```
  bt_callback_manager_register_callback(bt_callback_type_app_event,
(uint32_t)(MODULE_MASK_ GAP | MODULE_MASK_GATT | MODULE _MASK_SYSTEM |
MODULE_MASK_MM),(void *)bt_app_event_callback_demo);

  bt_status_t bt_app _event_ callback_demo(bt_msg_type_t msg, bt_status_t status,
void *buff)
  {
     switch (msg) {
       case BT_POWER_ON_CNF: {
          LOG_I(mesh_app, "BT_POWER_ON_CNF\n");
          bt_mesh_app_util_check_device_uuid(deviceUuid);

          bt_mesh_app_util_set_trl_mesh_mode(BTL_CTR_MESH_MODE_STANDBY);
```

```
            mesh_led_init();
            vendor_led_init();
            mesh_ota_init();
            mesh_timer_init();
            bt_gatts_set_max_mtu(69);

            if (!mesh_inited) {
               mesh_create_device();
             mesh_init();
          }else {
             bt_mesh_app_event_callback(BT_MESH_EVT_INIT_DONE, BT_MESH_SUCCESS,
NULL);
          }
          break;
        }
      }
    }
```

Declare a header for the upper-half of the composition data, excluding information for elements and models:

```
  uint8_t composition_data_header[10] = {
          0x94, 0x00, // cid
          0x1A, 0x00, // pid
          0x01, 0x00, // vid
          0x64, 0x00, // crpl
  #if RELAY_ENABLE
          BT_MESH_FEATURE_RELAY | BT_MESH_FEATURE_PROXY, 0x00, // features
  #else
          BT_MESH_FEATURE_PROXY, 0x00, // features
  #endif
  };
```

Set the composition data. Elements and models can then be added. As a provisioner, the most basic model is the Configuration Client model which allows recruiting a mesh-compliant device to be a member of a mesh network.

```
  bt_mesh_model_set_composition_data_header(composition_data_header, 10);
  bt_mesh_model_add_element(&element_index, BT_MESH_MODEL_ELEMENT_LOCATION_MAIN);

  //Adding a configuration server model
  bt_mesh_model_add_configuration_server(&g_model_handle_config_server, NULL);

  //Adding a health server model
  bt_mesh_model_add_health_server(&g_model_handle_health_server, element_index,
NULL);
```

The following example shows `meshinit()`:

```
bt_mesh_init_params_t *initparams = (bt_mesh_init_params_t
*)bt_mesh_app_util_alloc(sizeof(bt_mesh_init_params_t));

// init role
initparams->role = BT_MESH_ROLE_PROVISIONER;

// init config parameter
initparams->config = (bt_mesh_config_init_params_t
*)bt_mesh_app_util_alloc(sizeof(bt_mesh_config_init_params_t));

memset(initparams->config, 0, sizeof(bt_ mesh_config_ init_params_t));
memcpy(initparams->config->device _uuid, deviceUuid, BT_MESH_UUID_SIZE);
initparams->config->oob_info = 0;

initparams->config->uri = NULL;
initparams->config->default_ttl = 4;
initparams->friend = NULL;
bt_mesh_init(initparams);
bt_mesh_app_util_free((uint8_t *)(initparams->config));
bt_mesh_app_util_free((uint8_t *)initparams);
```

3.2.2 Using Pre-defined Models

The IoT SDK provides several different ready-to-use models for use. For example, basic controls can be switched on/off and the Lightness can be set for a simple Light device.

The following example shows the process of adding a Light Lightness client model.

```
bt_ mesh_model_lighting_add_lightness_client(&g_model_handle_light_lightne ss_
client, element_index, _lightness_client_msg_handler);
```

A typical scenario for a provisioner is to recruit mesh-compliant devices to form a proper mesh network. The process requires the provisioner to do the provisioning, configuring the provisioned device, such as providing the application key and binding device's models with the application key.

Therefore, the models that a provisioner typically needs are configuration client models.

```
bt_mesh_model_add_configuration_client(&g_model_handle_config_client,
element_index, _config_client_msg_handler);
```

Note: Please find pre-defined client models in headers named bt_mesh_model_*_client.

3.2.3 Creating a Customized Model

The model can be created if you cannot find the model in bt_mesh_model_*_client.h. Complete the following procedure to implement a customized model:

1) Define the opcode handlers. Define a table of handlers for incoming messages by creating an array of bt_mesh_access_opcode_handler_t. The table serves as a lookup table for handling the defined opcodes of incoming messages.

```
const bt_mesh_access_opcode_handler_t vendor_message_handler[] = {
    {{VENDOR_OPCODE_1, VENDOR_COMPANY_ID}, _vendorMsgHandler},
    {{VENDOR_OPCODE_2, VENDOR_COMPANY_ID}, _vendorMsgHandler}
};
```

2) Enter any necessary information and then add the new model. The model handle output parameter is used to identify the model. The model data in the composition data is also updated when a model is added.

```
// add vendor model bt_mesh_model_add_params_t model_params;
model_params.model_id = MESH_VENDOR_MODEL_ID(VENDOR_COMPANY_ID,VENDOR_MODEL_ID1);
model_params.element_index = element_index;
model_params.opcode_handlers = vendor_message_handler;
model_params.opcode_count = sizeof(vendor_message_handler);

model_params.publish_timeout_cb = NULL;
bt_mesh_model_add_model(&g_model_handle_vendor, &model_params);
```

3.2.4 Provisioning a Device

To create a mesh network, the provisioner starts a link with an unprovisioned device.

```
g_start_params = (bt_mesh_prov_provisioner_params_t
*)bt_mesh_app_util_alloc(sizeof(bt_mesh_prov_provisioner_params_t));
g_start_params->start.algorithm = BT_MESH_PROV_START_ALGORITHM_FIPS
_P256_ELLIPTIC_CURVE;
g_start_params->start.public_key = BT_MESH_PROV_START_PUBLIC_KEY_NO_OOB;
g_start_params->start.authentication_method =
BT_MESH_PROV_START_AUTHEN_METHOD_NO_OOB;

g_start_params->start.authentication_action = 0;
```

```
g_start_params->start.authentication_size = 0;
//fill in other information such as peer UUID, network key, IV index,
//attention_duration
bt_mesh_provision_invite_provisioning(uuid, &params);
```

When a device is provisioned, the provisioner must configure it. The provisioner must get the composition data of the peer device to know which models are supported by the peer device.

```
bt_mesh_model_configuration_client_get_composition_data(0xff,
bt_mesh_model_get_element_address(g_config_client_element_index), dst_addr,
bt_mesh_config_get_default_ttl(), BT_MESH_GLOBAL_PRIMARY_NETWORK_KEY_INDEX);
```

When a CONFIG_CLIENT_EVT_COMPOSITION_DATA_STATUS is received, the provisioner can provide the peer device with an application key and bind the key to the necessary models so that the models can decrypt access messages.

```
//Add an application key
bt_mesh_model_configuration_client_add_app_key( appkeyIdx, netkeyIdx, appkey,
bt_mesh_model_get_element_address(g_config_client_element_index), dst_addr,
bt_mesh_config_get_default_ttl(),
BT_MESH_GLOBAL_PRIMARY_NETWORK_KEY_INDEX);

//Bind model with the application key
bt_mesh_model_configuration_client_bind_model_app( element_addr, appkeyidx,
model_id,
bt_mesh_model_get_element_address(g_config_client_element_index), dst_addr,
bt_mesh_config_get_default_ttl(), BT_MESH_GLOBAL_PRIMARY_NETWORK_KEY_INDEX);
```

3.2.5 Sending Access Messages to a Provisioned Device

Access messages can be sent when the provisioned device is configured. One method is to send messages with APIs defined in the IoT SDK. The corresponding APIs for sending messages are pre-defined if the model is created with a pre-defined API.

```
bt_mesh_access_message_tx_meta_t tx_meta = {0};

tx_meta.dst_addr.value = dst_addr;
tx_meta.dst_addr.type = bt_mesh_utils_get_addr_type(dst_addr);
tx_meta.appkey_index = g_appkey_index;

tx_ meta.ttl = ttl;
tx_ meta.dst_addr.virtual_uuid = NULL;

bt_mesh_status_t ret =
bt_mesh_model_generic_onoff_client_get(g_model_handle_onoff_client, &tx_meta);
```

If you want to send a vendor-customized access message, the bt_mesh_send_packet() API can be used as shown in the following example:

```
// set packet buffer, please refer to Mesh Profile Specification 3.7.3 uint16_t i
= 0;
uint8_t *payload = malloc(param.data_len);

payload[0] = opcode;
payload[1] = company_id & 0x00FF;
payload[2] = (company_id & 0xFF00) >> 8;
for(i = 0; i < datalen ; i++) {
    payload[3+i] = i;
}
param.data = payload;
param.security.appidx = g_appkey_index;
param.security.device_key = NULL;

bt_mesh_status_t ret = bt_mesh_send_packet(&param);
```

3.3 Writing an Application of Provisionee

3.3.1 Initialization

The following example shows when the mesh-related initialization can start.

The initialization flow of the provisionee for the BT section is the same as the provisioner, but the data of the mesh initialization structure is different.

```
initparams->role = BT_MESH_ROLE_PROVISIONEE;

// init provision parameter
initparams->provisionee = (bt_mesh_prov_provisionee_params_t
*)bt_mesh_app_util_alloc(sizeof(bt_mesh_prov_provisionee_params_t));

initparams->provisionee->cap.number_of_elements =
bt_mesh_model_get_element_count();

initparams->provisionee->cap.algorithms =
BT_MESH_PROV_CAPABILITY_ALGORITHM_FIPS_P256_ELLIPTIC_CURVE; // bit 0: P-256,
```

bit 1~15: RFU

```
initparams->provisionee->cap.public_key_type    =
BT_MESH_PROV_CAPABILITY_OOB_PUBLIC_KEY_TYPE_INBAND;
initparams->provisionee->cap.static_oob_type    =
BT_MESH_PROV_CAPABILITY_OOB_STATIC_TYPE_SUPPORTED;
initparams->provisionee->cap.output_oob_size = 0x00;
initparams->provisionee->cap.output_oob_action = 0x0000;
initparams->provisionee->cap.input_oob_size = 0x00;
initparams->provisionee->cap.input_oob_action = 0x0000;
```

3.3.2 Adding Models

The IoT SDK provides several different ready-to-use models for use. For example, basic controls can be switched on/off and the Lightness can be set for a simple Light device.

The following example shows the process of adding a Light Lightness server model.

```
bt_mesh_model_lighting_add_lightness_setup_server(&g_model_handle_lightness_
server, &element_count, &element_list, element_index, light_server_msg_handler,
NULL);
```

Note: Headers named bt_mesh_model_*_server.h. contain the pre-defined server models.

Otherwise, a customized client model can be created using the same method as a customized client model.

According to SIG Mesh Model specification, a server model can extend one or more server models. Model extension is already implemented in the IOT SDK; adding the top-most server model is enough for you. For example, the Light Lightness Server model extends the Generic Power OnOff Server model and the Generic Level Server model. Calling the `bt_mesh_model_lighting_add_lightness_setup_server()` API automatically creates a Generic Power OnOff Server model and the Generic Level Server model on the device.

The following example is for a Light CTL server. The default values for some fields must first be assigned.

```
gCTL_server->range_min = 0x320;
gCTL_server->range_max = 0x4E20;
gCTL_server->default_temperature = 0x320;     // default value for
GENERIC_ON_POWERUP_RESTORE
gCTL_server->default_delta_uv = 0;            // default value for
GENERIC_ON_POWERUP_RESTORE
gCTL_server->target_ctl_temperature= 0x2000;             // last known value for
GNERIC_ON_POWERUP_RESTORE
gCTL_server->target_ctl_delta_uv = -100;                 // last known value for
GNERIC_ON_POWERUP_RESTORE
gCTL_server->lightness_server.target_lightness = 0xabcd; // last known value for
GNERIC_ON_POWERUP_RESTORE
gCTL_server->lightness_server.last_lightness = 0x1234;
gCTL_server->lightness_server.default_lightness = 0x0000;// default for
GNERIC_ON_POWERUP_RESTORE
gCTL_server->lightness_server.range_min = 0x0001;
gCTL_server->lightness_server.range_max = 0xffff;
gCTL_server->lightness_server.onoff_server.target_on_off = 1; // last known value
for GENERIC_ON_POWERUP_RESTORE
gCTL_server->lightness_server.onpowerup_server.on_power_up =
GENERIC_ON_POWERUP_RESTORE;

gCTL_server->lightness_server.TID = 0xF0;                //default value for TID
gCTL_server->lightness_server.level_server.TID = 0xF0;
gCTL_server->lightness_server.dtt_server = gTransition_timer;

gCTL_server->element_index = 0xFFFF;
gCTL_server->TID = 0xF0; //default value for TID

gCTL_temperature_server->element_index = 0xFFFF;
gCTL_temperature_server->TID = 0xF0;
```

The CTL setup server model can then be added:

```
bt_mesh_model_lighting_add_ctl_setup_server(&g_model_handle_ctl_server,
&element_count, &element_list, element_index, light_ctl_server_msg_handler,
NULL);

gCTL_server->element_index = element_index;
if(element_list != NULL && element_list[0] == element_index)
{
   gCTL_temperature_server->element_index = element_list[1];
}
```

Note: Server models such as CTL server and HSL server models are scattered across more than one element. You do not need to add elements to accommodate these models; they can simply call the original CTL/HSL model and complete the element index based on the output parameter as shown in the previous example.

3.3.3 Handling Mesh Messages

After models are created, the device is ready to receive mesh messages if the appropriate security information exists. From the viewpoint of a server model, a typical scenario is to receive various messages and reply to them with the status.

The following example shows a message handler manages the BT_MESH_MODEL_GENERIC_ONOFF_SET message. The message handler is the parameter that is added when the model is created.

```
static void _generic_server_msg_handler(uint16_t model_handle, const
bt_mesh_access_message_rx_t *msg, const void *arg)
{
   switch (msg->opcode.opcode) {
      case BT_MESH_MODEL_GENERIC_ONOFF_SET: {
        _generic_onoff_set_handler(model_handle, msg, arg, true);
        break;
   }
}
```

Note: As defined in the Mesh Model specification, a state can be bound to another state. State binding must be handled in application code.

4 Debugging and Porting Layer

4.1 Debugging

The IoT SDK provides two ways to debug the application:

- System log — provides the Bluetooth mesh information.
- HCI log — provides the HCI communication log between Bluetooth host and controller.

4.1.1 System Log

The Bluetooth mesh stack calls `bt_mesh_debug_log()` in the `bt_mesh_debug_layer.c` source file to provide the host information. The application can set a filter or disable logging in this function. The application can also output these logs to any UART or USB port.

4.1.2 HCI Log

Please refer to a different developers guide < MT793X IoT SDK Bluetooth Developer's Guide.pdf>.

4.2 Porting Layer

The Bluetooth mesh stack has the flexibility to integrate with third-party RTOS APIs. These APIs can be implemented in the source file `bt_mesh_os_layer_api.c`.

Exhibit 1 Terms and Conditions

Your access to and use of this document and the information contained herein (collectively this "Document") is subject to your (including the corporation or other legal entity you represent, collectively "You") acceptance of the terms and conditions set forth below ("T&C"). By using, accessing or downloading this Document, You are accepting the T&C and agree to be bound by the T&C. If You don't agree to the T&C, You may not use this Document and shall immediately destroy any copy thereof.

This Document contains information that is confidential and proprietary to MediaTek Inc. and/or its affiliates (collectively "MediaTek") or its licensors and is provided solely for Your internal use with MediaTek's chipset(s) described in this Document and shall not be used for any other purposes (including but not limited to identifying or providing evidence to support any potential patent infringement claim against MediaTek or any of MediaTek's suppliers and/or direct or indirect customers). Unauthorized use or disclosure of the information contained herein is prohibited. You agree to indemnify MediaTek for any loss or damages suffered by MediaTek for Your unauthorized use or disclosure of this Document, in whole or in part.

MediaTek and its licensors retain titles and all ownership rights in and to this Document and no license (express or implied, by estoppels or otherwise) to any intellectual propriety rights is granted hereunder. This Document is subject to change without further notification. MediaTek does not assume any responsibility arising out of or in connection with any use of, or reliance on, this Document, and specifically disclaims any and all liability, including, without limitation, consequential or incidental damages.

THIS DOCUMENT AND ANY OTHER MATERIALS OR TECHNICAL SUPPORT PROVIDED BY MEDIATEK IN CONNECTION WITH THIS DOCUMENT, IF ANY, ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE. MEDIATEK SPECIFICALLY DISCLAIMS ALL WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, COMPLETENESS OR ACCURACY AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MEDIATEK SHALL NOT BE RESPONSIBLE FOR ANY MEDIATEK DELIVERABLES MADE TO MEET YOUR SPECIFICATIONS OR TO CONFORM TO A PARTICULAR STANDARD OR OPEN FORUM.

Without limiting the generality of the foregoing, MediaTek makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does MediaTek assume any liability arising out of the application or use of any product, circuit or software. You agree that You are solely responsible for the designing, validating and testing Your product incorporating MediaTek's product and ensure such product meets applicable standards and any safety, security or other requirements.

The above T&C and all acts in connection with the T&C or this Document shall be governed, construed and interpreted in accordance with the laws of Taiwan, without giving effect to the principles of conflicts of law.