GR533x Mesh Light Lightness Models Example Application

Version: 1.1

Release Date: 2023-11-06

Shenzhen Goodix Technology Co., Ltd.

Trademarks and Permissions

GOODiX and other Goodix trademarks are trademarks of Shenzhen Goodix Technology Co., Ltd. All other trademarks and trade names mentioned in this document are the property of their respective holders.

Disclaimer

Information contained in this document is intended for your convenience only and is subject to change without prior notice. It is your responsibility to ensure its application complies with technical specifications.

Shenzhen Goodix Technology Co., Ltd. (hereafter referred to as “Goodix”) makes no representation or guarantee for this information, express or implied, oral or written, statutory or otherwise, including but not limited to representation or guarantee for its application, quality, performance, merchantability or fitness for a particular purpose. Goodix shall assume no responsibility for this information and relevant consequences arising out of the use of such information.

Without written consent of Goodix, it is prohibited to use Goodix products as critical components in any life support system. Under the protection of Goodix intellectual property rights, no license may be transferred implicitly or by any other means.

Shenzhen Goodix Technology Co., Ltd.

Headquarters: Floor 12-13, Phase B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828 Zip Code: 518000

Website: www.goodix.com

Preface

Purpose

This document provides an overview of the operation, verification, and critical code of the Light Lightness Models examples in GR533x Software Development Kit (SDK), to help users quickly get started with secondary development.

Audience

This document is intended for:

- Device user
- Developer
- Teste engineer
- Technical support engineer

Release Notes

This document is the second release of *GR533x Mesh Light Lightness Models Example Application*, corresponding to GR533x System-on-Chip (SoC) series.

Revision History

Version	Date	Description
1.0	2023-10-18	Initial release
1.1	2023-11-06	Updated the approache for obtaining GRUart.

Contents

1 Introduction

To ensure the compatibility for message exchange between different kinds of Bluetooth Mesh devices, Bluetooth Special Interest Group (Bluetooth SIG) defines a series of generic and standard models for Bluetooth Mesh applications, including Generics, Sensors, Time and Scenes, and Lighting, which enable Bluetooth Mesh devices to control a peer Bluetooth device or obtain the device information, facilitating Mesh applications in different scenarios.

This document elaborates on how to use standard models and develop applications with these models by taking the Light Lightness (LLN) Models in Lighting Models for example.

Before getting started, you can refer to the following documents.

Table 1-1 Reference documents

Name	Description
GR533x Developer Guide	Introduces the software/hardware and quick start guide of GR533x System-on-Chips (SoCs).
Bluetooth Core Spec	Offers official Bluetooth standards and core specification from Bluetooth SIG.
Mesh Networking Specifications	Offers Bluetooth official Mesh profiles and specifications. Available at https://www.bluetooth.com/specifications/mesh-specifications/.
J-Link/J-Trace User Guide	Provides J-Link operational instructions. Available at https://www.segger.com/downloads/jlink/UM08001_JLink.pdf.
Keil User Guide	Offers detailed Keil operational instructions. Available at https://www.keil.com/support/man/docs/uv4/.

2 Overview of LLN Models

Goodix provides LLN Models for your reference, to demonstrate how to implement standard models.

LLN Models include:

- LLN Client model: It sends messages to a server model to obtain or change the states (including on/off state, lightness level, lightness default, and lightness range) of the server model.
- Two server models:
  - LLN Server Model: It receives messages from the client model and responds with or changes the on/off state and lightness level according to requests from the client model.
  - Light Lightness Setup (LLNS) Server Model: It receives messages from the client model and responds with or changes the lightness default and lightness range according to requests from the client model.

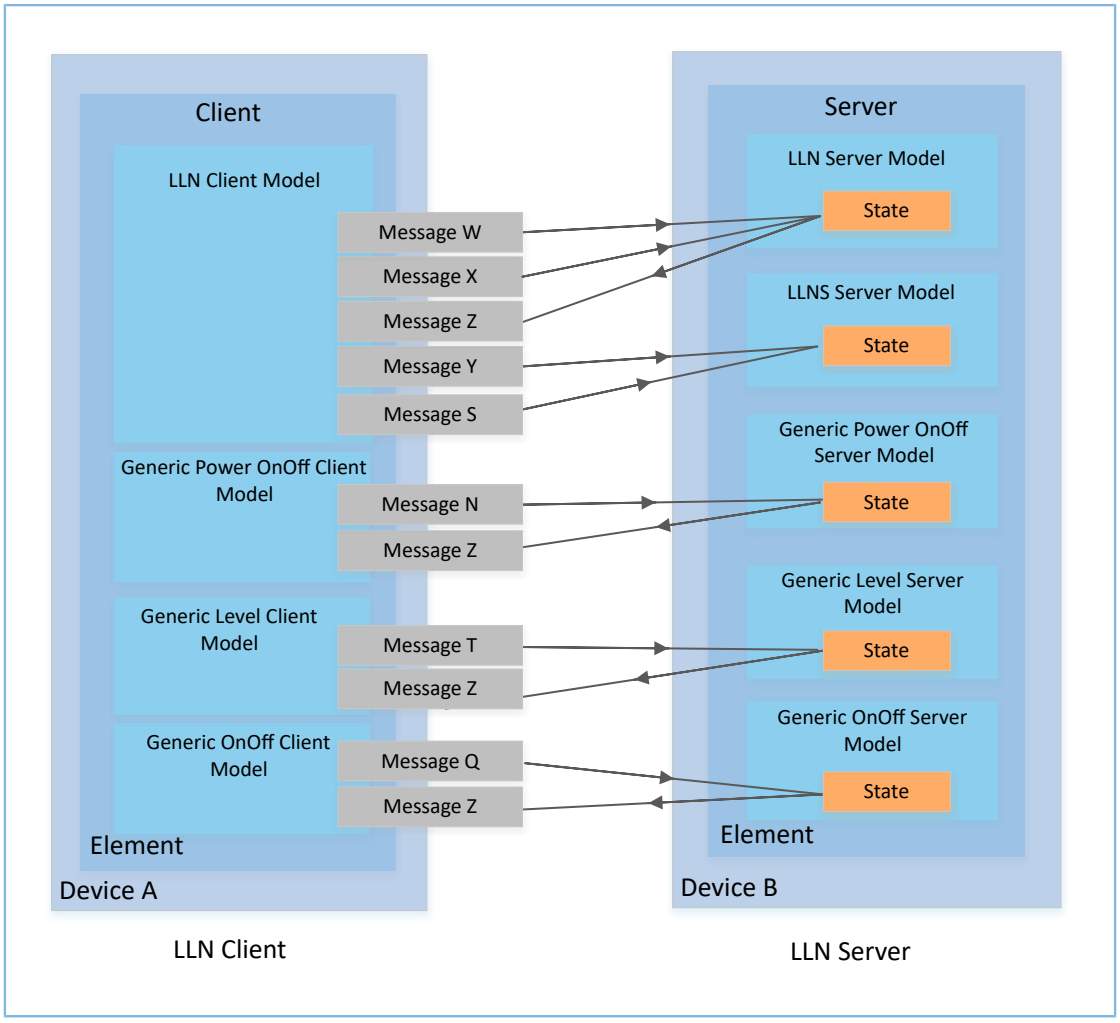

Figure 2-1 Message exchange between LLN Client Model and LLN server models

As shown in Figure 2-1, a complete LLN Server element includes:

- Two main server models:

  - LLN Server Model
  - LLNS Server Model
- Three extended server models:
  - Generic Power OnOff Server Model
  - Generic Level Server Model
  - Generic OnOff Server Model

The main server models inherit behaviors of the extended server models, and the states of the two types of models are bound together.

To implement message exchange with the LLN Server, the LLN Client shall contain client models which correspond to certain server models, including:

- LLN Client Model: corresponding to LLN Server Model and LLNS Server Model
- Generic Power OnOff Client Model: corresponding to Generic Power OnOff Server Model
- Generic Level Client Model: corresponding to Generic Level Server Model
- Generic OnOff Client Model: corresponding to Generic OnOff Server Model

2.1 Published Message Type

The published message types supported by the LLN Client Model are listed as follows.

Table 2-1 Published message type supported by LLN Client Model

Message Type	Description
LIGHT_LIGHTNESS_OPCODE_GET	Obtain the current Lightness Actual state of the LLN Server and wait for response.
LIGHT_LIGHTNESS_OPCODE_SET	Set the current Lightness Actual state of the LLN Server and wait for response.
LIGHT_LIGHTNESS_OPCODE_SET_UNACKNOWLEDGED	Set the current Lightness Actual state of the LLN server without waiting for response.
LIGHT_LIGHTNESS_LINEAR_OPCODE_GET	Obtain the current Lightness Linear state of the LLN Server and wait for response.
LIGHT_LIGHTNESS_LINEAR_OPCODE_SET	Set the current Lightness Linear state of the LLN Server and wait for response.
LIGHT_LIGHTNESS_LINEAR_OPCODE_SET_UNACKNOWLEDGED	Set the current Lightness Linear state of the LLN server without waiting for response.
LIGHT_LIGHTNESS_LAST_OPCODE_GET	Obtain the Lightness Last state of the LLN Server and wait for response.

Message Type	Description
LIGHT_LIGHTNESS_DEFAULT_OPCODE_GET	Obtain the Lightness Default state of the LLN Server and wait for response.
LIGHT_LIGHTNESS_RANGE_OPCODE_GET	Obtain the Lightness Range state of the LLN Server and wait for response.
LIGHT_LIGHTNESS_DEFAULT_OPCODE_SET	Set the current Lightness Default state of the LLN Server and wait for response.
LIGHT_LIGHTNESS_DEFAULT_OPCODE_SET_UNACKNOWLEDGED	Set the current Lightness Default state of the LLN Server without waiting for response.
LIGHT_LIGHTNESS_RANGE_OPCODE_SET	Set the current Lightness Range state of the LLN Server and wait for response.
LIGHT_LIGHTNESS_RANGE_OPCODE_SET_UNACKNOWLEDGED	Set the current Lightness Range state of the LLN Server without waiting for response.

The published message types supported by the LLN Server Model are listed as follows.

Table 2-2 Published message type supported by LLN Server Model

Message Type	Description
LIGHT_LIGHTNESS_OPCODE_STATUS	Publish the current Lightness Actual state of the LLN Server.
LIGHT_LIGHTNESS_LINEAR_OPCODE_STATUS	Publish the current Lightness Linear state of the LLN Server.
LIGHT_LIGHTNESS_LAST_OPCODE_STATUS	Publish the Lightness Last state of the LLN Server.
LIGHT_LIGHTNESS_DEFAULT_OPCODE_STATUS	Publish the Lightness Default state of the LLN Server.
LIGHT_LIGHTNESS_RANGE_OPCODE_STATUS	Publish the Lightness Range state of the LLN Server.

Note:

The LLNS Server Model does not support publishing messages; it only receives and processes messages sent by the LLN Client Model. Messages shall be published by the LLN Server Model.

3 Example Running

This chapter introduces how to run the LLN Models example projects and test/verify the running results.

LLN Models example projects include:

- LLN Client Model example project: in SDK_Folder\projects\mesh\SIG\mesh_app_light_lightness_client
- LLN Server Model example project: in SDK_Folder\projects\mesh\SIG\mesh_app_light_lightness_server

---

Note:

SDK_Folder is the root directory of GR533x Software Development Kit (SDK).

3.1 Preparation

Perform the following tasks before running the LLN Models example projects.

- **Hardware preparation**

Table 3-1 Hardware preparation

Name	Description
J-Link debug probe	JTAG emulator launched by SEGGER. For more information, visit www.segger.com/products/debug-probes/j-link/.
Development board	GR5331 Starter Kit Board (GR5331 SK Board) (2 boards in total)
Connection cable	USB Type-C cable

- **Software preparation**

Table 3-2 Software preparation

Name	Description
Windows	Windows 7/Windows 10
J-Link driver	A J-Link driver. Available at www.segger.com/downloads/jlink/.
Keil MDK5	An integrated development environment (IDE). Available at www.keil.com/download/product/.
GRMesh (Android)	A Mesh debugging tool.
GRUart	A serial port debugging tool. Available at www.goodix.com/en/download?objectId=43&objectType=software.

3.2 Hardware Connection

Use two GR5331 SK Boards to serve as the LLN Client and the LLN Server respectively.

Connect the GR5331 SK Boards to a PC with USB Type-C cables.

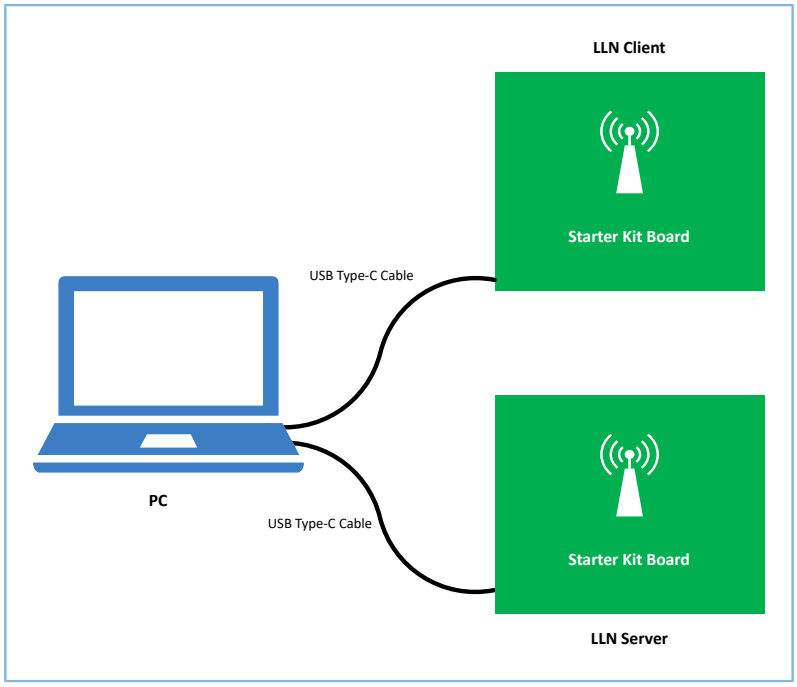

Figure 3-1 Hardware connection

3.3 Firmware Download

Compile projects mesh_app_light_lightness_client and mesh_app_light_lightness_server in Keil, and download the generated target files to the two GR5331 SK Boards.

3.4 Firmware Running

Press **S9** (**RESET** button) on the GR5331 SK Board to run the firmware after firmware download completes.

3.5 Configuration of Serial Port Tool

Start GRUart, and configure the parameters as follows.

Table 3-3 Configuring parameters on GRUart

PortName	BaudRate	DataBits	Parity	StopBits	Flow Control
Select on demand	115200	8	None	1	Uncheck

The figure below shows the interface of GRUart after configuration completes.

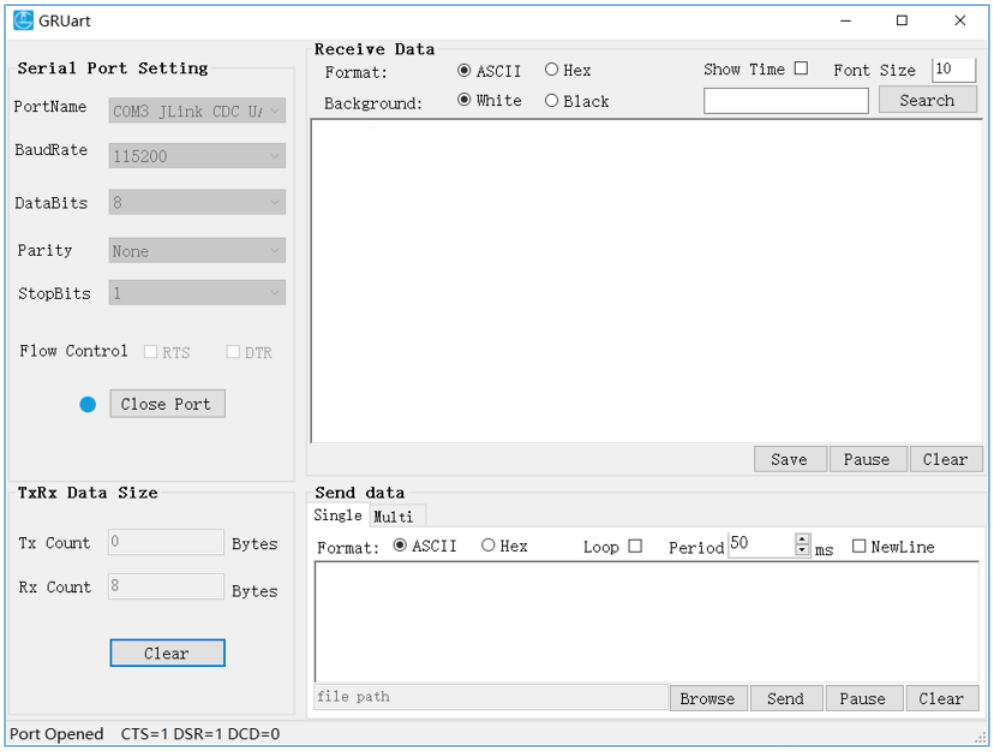

Figure 3-2 GRUart configuration

3.6 Test and Verification

After the previous preparations are ready, you can test and verify communication between the LLN Models according to the following steps.

1. Configure the LLN Client and the LLN Server through GRMesh.
2. Verify message interactions between the LLN Client and the LLN Server by checking the logs printed on GRUart.

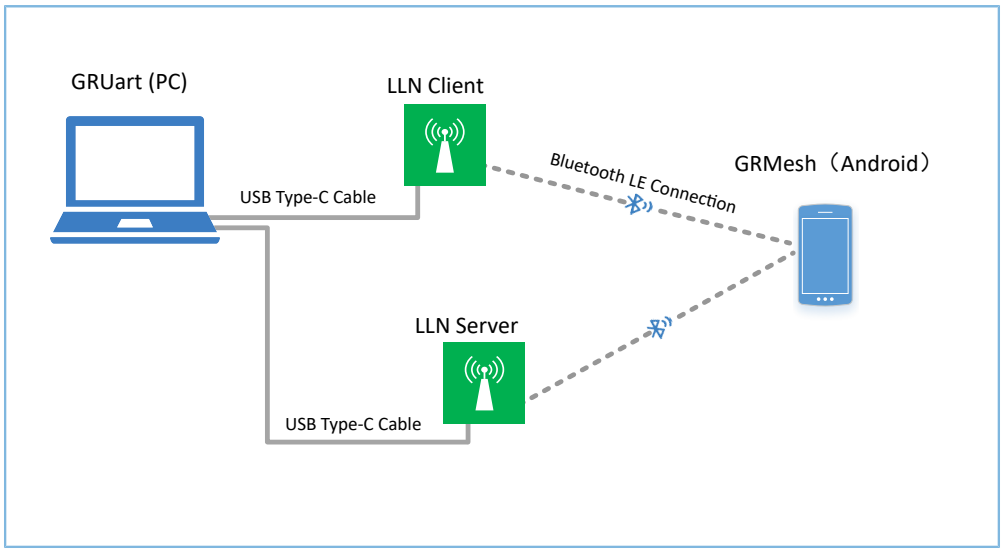

Figure 3-3 LLN Models test diagram

3.6.1 Device Configuration through GRMesh

Configure the LLN Client and the LLN Server through GRMesh as follows:

1. Power on the LLN Client (Goodix_LLNC) and the LLN Server (Goodix_LLNS).
2. Provision Goodix_LLNC on GRMesh.

   (1). To add a new device, tap **+** in the upper-right corner on the home page of GRMesh to start scanning. A device with the advertising name of Goodix_LLNC is discovered.

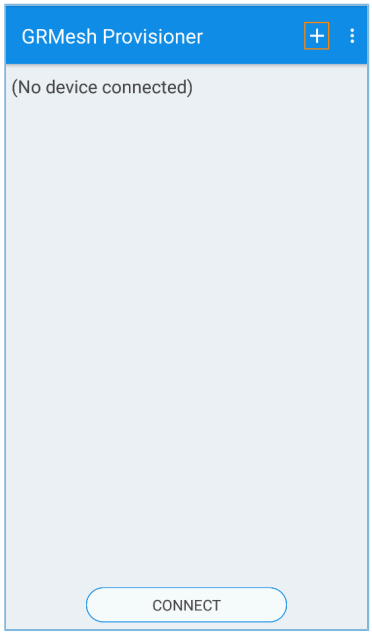

Figure 3-4 To add a new device

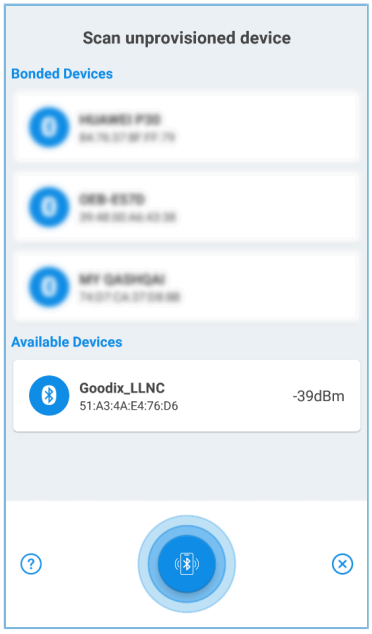

Figure 3-5 Discovering a new device

   (2). Tap **Goodix_LLNC** > **IDENTIFY** > **PROVISION**.

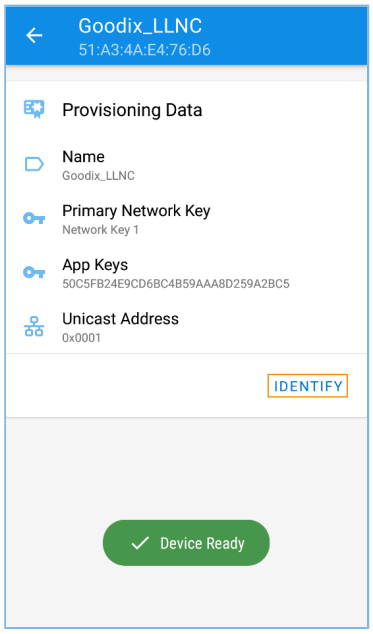

Figure 3-6 Tapping **IDENTIFY**

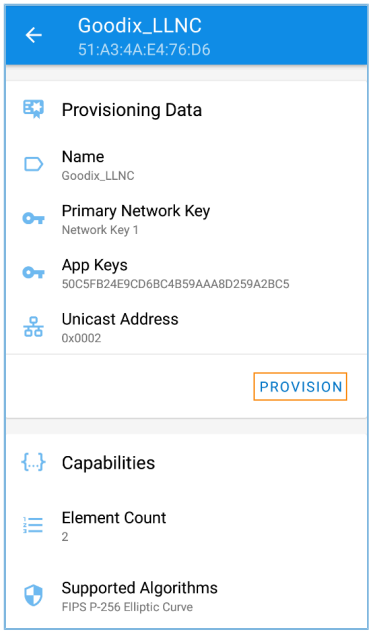

Figure 3-7 Tapping **PROVISION**

(3). On the **Select OOB Type** pane, select **No OOB** from the drop-down list and then tap **OK** to complete provisioning of the Goodix_LLNC. Then the LLN Client is added to the Mesh network.

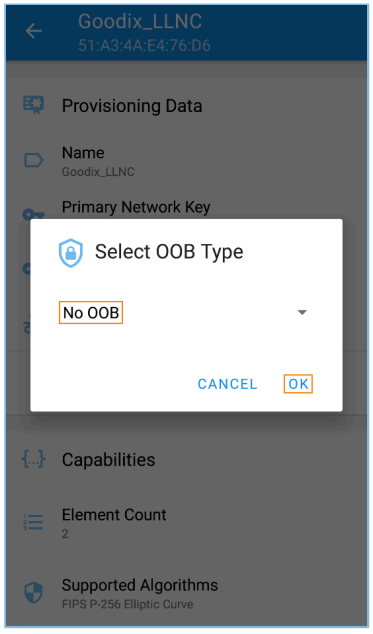

Figure 3-8 Tapping **OK**

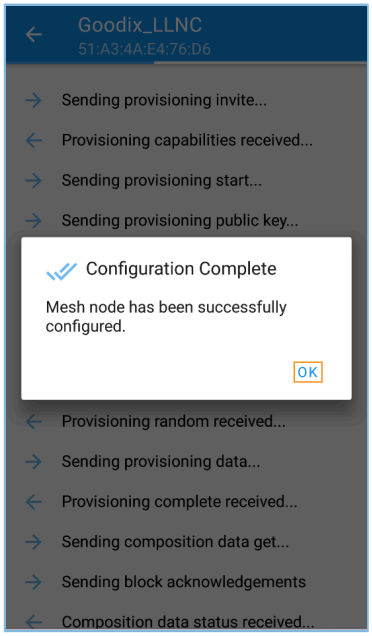

Figure 3-9 Completing provisioning

3. On the **GRMesh Provisioner** interface, tap **Goodix_LLNC** > **Element:0x0002** > **Light Lightness Client Model**, to configure the LLN Client Model of the first element of Goodix_LLNC.

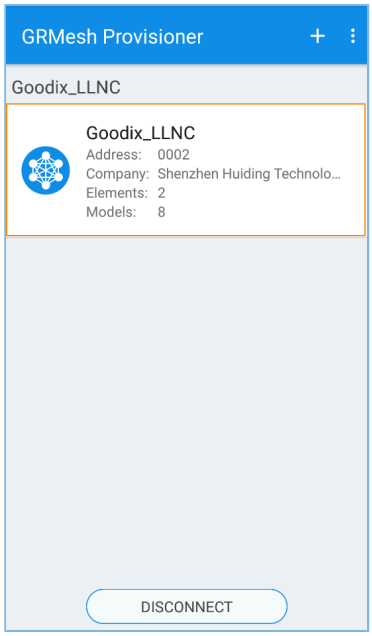

Figure 3-10 Configuring Goodix_LLNC

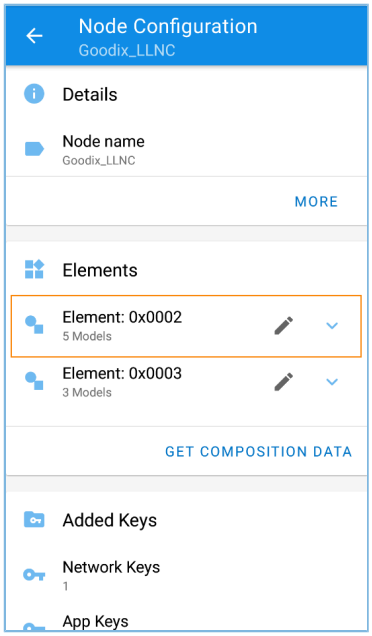

Figure 3-11 Tapping **Vendor Model**

On the **Light Lightness Client** interface, configure the **Bound App Keys** of this model as **index 0**, and the unicast address of **Publish Address** as **0x0004** (corresponding to the address of the first element of Goodix_LLNS).

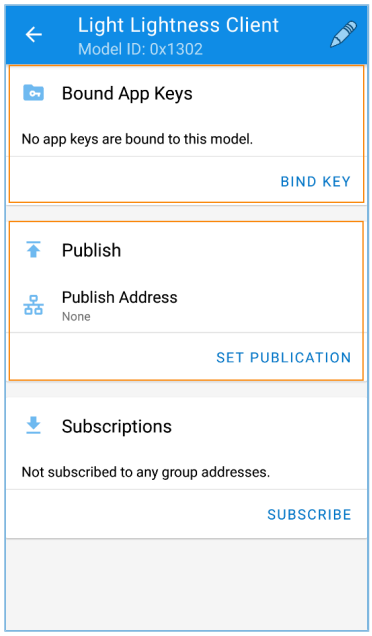

Figure 3-12 **Vendor Model** interface

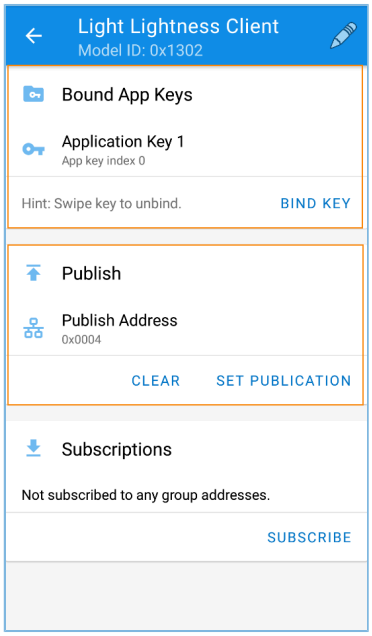

Figure 3-13 Configuring **Bound App Keys** and **Publish Address**

4. Refer to Step 2 and Step 3 to configure the Generic OnOff Client Model and the Generic Level Client Model of the first element of Goodix_LLNC.

5. Refer to Step 2 and Step 3 to provision Goodix_LLNS, and configure the **Publish Address** (unicast address) of the LLN Server Model of the first element as **0x0002** (corresponding to the address of the first element of Goodix_LLNC).

6. Set **Bound App Keys** of the Generic OnOff Server Model, the Generic Level Server Model, and the LLNS Server Model of the first element of Goodix_LLNS to **index 0**.

7. After provisioning of the LLN Client and the LLN Server, you can view device information in the **GRMesh Provisioner** interface, as shown below.

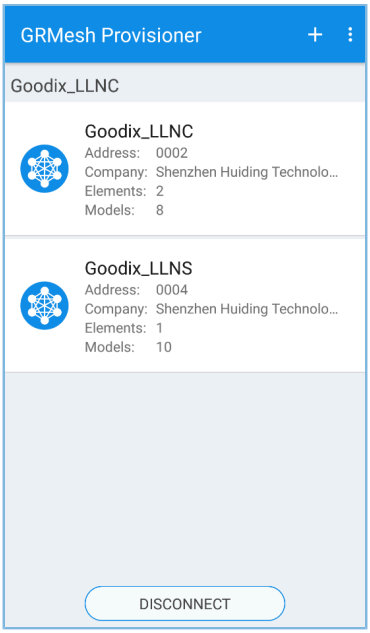

Figure 3-14 Completing provisioning

3.6.2 Verification of LLN Models through GRUart

Test communication interaction between the LLN Client and the LLN Server through GRUart as follows:

1. Open two GRUart windows: one for LLN Client and the other for LLN Server.

2. Input **00010490015400** to GRUart (LLN Client).

   (1). GRUart (LLN Client) sends the **LIGHT_LIGHTNESS_OPCODE_SET** message (with the `Lightness Actual` command) to set Lightness Actual (400, 0x0190), transition time (2000 ms, 0x0054), and delay time (default: 0 ms).

   (2). The LLN Server sets the transition time and delay time after receiving the message and immediately returns “LIGHT_LIGHTNESS_OPCODE_STATUS” to the LLN Client.

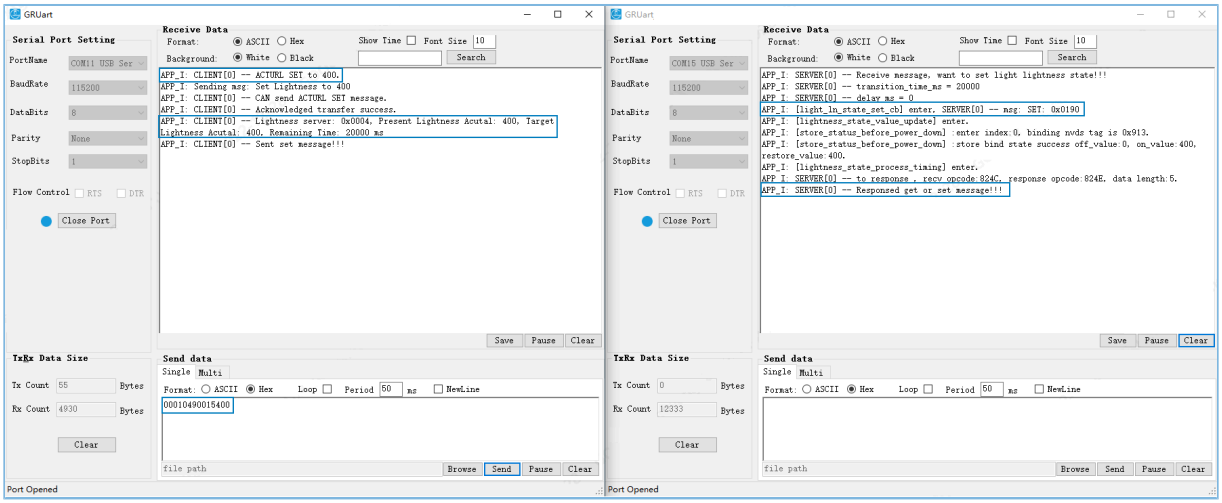

Figure 3-15 Message exchange between LLN Client (left) and LLN Server (right) - 1

If transition timeout (longer than 2000 ms) occurs, the LLN Server sets **Lightness Actual State** to the corresponding value and publishes the state to the LLN Client, as shown below.

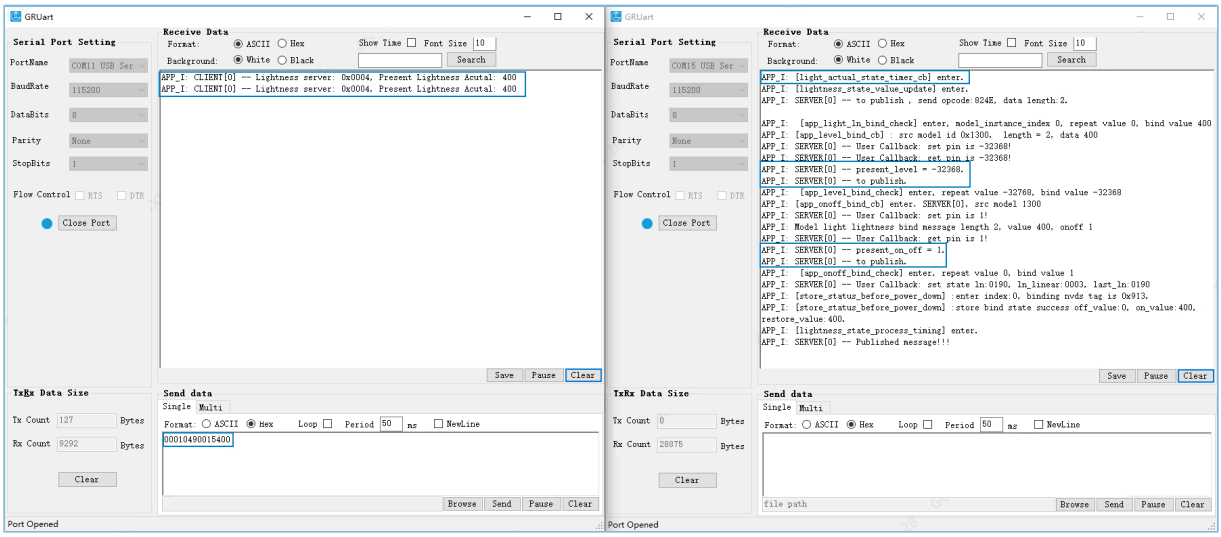

Figure 3-16 Message exchange between LLN Client (left) and LLN Server (right) - 2

3. Input **000B04640000F0** to GRUart (LLN Client).

   (1). The LLN Client sends the **LIGHT_LIGHTNESS_RANGE_OPCODE_SET** message (with the `Lightness range` command) to set Range Min (100, 0x0064) and Range Max (61440, 0xF000).

   (2). The LLN Server sets the Lightness Range to the corresponding value and responds to the message through a server model after receiving the message.

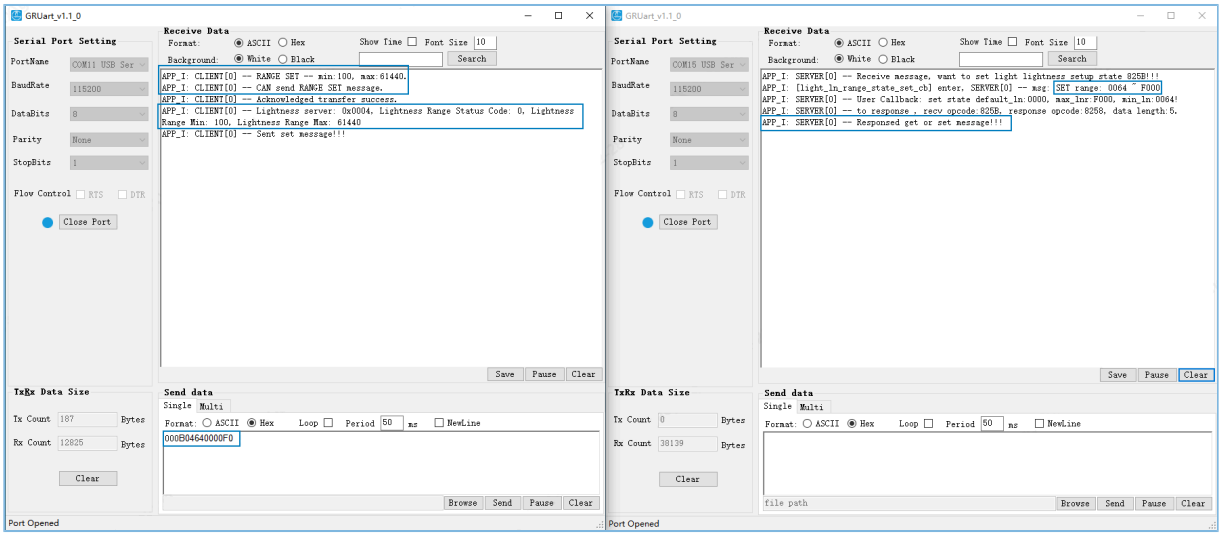

Figure 3-17 Message exchange between LLN Client (left) and LLN Server (right) - 3

4 Application Details

This chapter introduces the project directory of the LLN Models examples and the implementation code of critical functionalities.

4.1 Project Directory

GR533x SDK provides source code and project files of the LLN Client Model and LLN Server Model examples, to help users implement custom LLN Models by referring to the example projects.

4.1.1 LLN Client Model Project

The source code and project file of the LLN Client Model example are in SDK_Folder\projects\mesh\SIG\mesh_app_light_lightness_client, and the project file is in the Keil_5 folder.

Open *mesh_app_light_lightness_client.uvprojx* in Keil, to view the project directory structure of the LLN Client Model example. Details of the files are listed below.

Table 4-1 File description of mesh_app_light_lightness_client

Group	File	Description
gr_models	light_lightness_client.c	Implement LLN Client Model.
	Generic_level_client.c	Implement Generic Level Client Model.
	Generic_onoff_client.c	Implement Generic OnOff Client Model.
	Generic_power_onoff_client.c	Implement Generic Power OnOff Client Model.
user_callback	user_gap_callback.c	Obtain connection and disconnection events.
	user_mesh_callback.c	Obtain provisioning events.
user_platform	user_periph_setup.c	Configure App logs, serial port parameters, and device address.
user_app	main.c	Contain the main() function.
	user_app.c	Register client models and process user logics.
	App_light_lightness_client.c	Implement application initialization and registration for LLN Client Model.
	App_level_client.c	Implement application initialization and registration for Generic Level Client Model.
	App_onoff_client.c	Implement application initialization and registration for Generic OnOff Client Model.

4.1.2 LLN Server Model Project

The source code and project file of the LLN Server Model example are in SDK_Folder\projects\mesh\SIG\mesh_app_light_lightness_server, and the project file is in the Keil_5 folder.

Open *mesh_app_light_lightness_server.uvprojx* in Keil, to view the project directory structure of the LLN Server Model example. Details of the files are listed below.

Table 4-2 File description of mesh_app_light_lightness_server

Group	File	Description
gr_models	light_lightness_server.c	Implement LLN Server Model.
	light_lightness_setup_server.c	Implement LLNS Server Model.
	generic_level_server.c	Implement Generic Level Server Model.
	generic_onoff_server.c	Implement Generic OnOff Server Model.
	generic_power_onoff_server.c	Implement Generic Power OnOff Server Model.
	generic_power_onoff_setup_server.c	Implement Generic Power OnOff Setup Server Model.
	generic_power_onoff_behavior.c	Implement the Generic Power OnOff method.
	generic_default_transition_time_server.c	Implement Generic Default Transition Time Server Model.
	scene_server.c	Implement Scene Server Model.
	scene_setup_server.c	Implement Scene Setup Server Model.
user_callback	user_gap_callback.c	Obtain connection and disconnection events.
	user_mesh_callback.c	Obtain provisioning events.
user_platform	user_periph_setup.c	Configure App logs, serial port parameters, and device address.
user_app	main.c	Contain the main() function.
	user_app.c	Register server models and process user logics.
	app_light_lightness_server.c	Implement application initialization, registration, and logic processing for LLN Server Model.
	app_light_lightness_setup_server.c	Implement application initialization, registration, and logic processing for LLNS Server Model.
	app_level_server.c	Implement application initialization, registration, and logic processing for Generic Level Server Model.
	app_onoff_server.c	Implement application initialization, registration, and logic processing for Generic OnOff Server Model.
	app_power_onoff_server.c	Implement application initialization, registration, and logic processing for Generic Power OnOff Server Model.
	app_power_onoff_setup_server.c	Implement application initialization, registration, and logic processing for Generic Power OnOff Setup Server Model.
	app_scene_server.c	Implement application initialization, registration, and logic processing for Scene Server Model.
	app_scene_setup_server.c	Implement application initialization, registration, and logic processing for Scene Setup Server Model.
	custom_config.h	Common configurations for application projects
	mesh_stack_config.h	Configurations related to Mesh protocols

4.2 Code Description

4.2.1 Message Processing of LLN Models

This section demonstrates how to implement messages processing for standard models by introducing the implementation code for message processing of LLN Models.

4.2.1.1 LLN Client Model

The example code is in SDK_Folder\components\mesh\models\SIG\Lighting\light_lightness\src\light_lightness_client.c.

4.2.1.1.1 Processable Message List

The Mesh messages to be received and processed by the LLN Client Model are listed in the processable message list light_ln_client_opcode_list.

```
static const uint16_t light_ln_client_opcode_list[] =
{
    LIGHT_LIGHTNESS_OPCODE_STATUS,
    LIGHT_LIGHTNESS_LINEAR_OPCODE_STATUS,
    LIGHT_LIGHTNESS_LAST_OPCODE_STATUS,
    LIGHT_LIGHTNESS_DEFAULT_OPCODE_STATUS,
    LIGHT_LIGHTNESS_RANGE_OPCODE_STATUS,
};
```

4.2.1.1.2 Callback Function Handling List

The callback function list light_ln_client_msg_cb consists of three parts:

- cb_rx: message callback function received by the LLN Client Model
- cb_sent: callback function published or returned by the LLN Client Model to indicate message state
- cb_publish_period: callback function when the periodic publish state of the LLN Client Model changes

```
static const mesh_model_cb_t light_ln_client_msg_cb =
{
    .cb_rx              = light_ln_client_rx_cb,
    .cb_sent            = light_ln_client_sent_cb,
    .cb_publish_period  = NULL,
};
```

The code snippet for processing of Mesh messages received by light_ln_client_rx_cb() is as follows.

```
static void light_ln_client_rx_cb(mesh_model_msg_ind_t *p_model_msg, void *p_args)
{
    uint16_t company_opcode = p_model_msg->opcode.company_opcode;
    uint8_t index = 0;
    switch(company_opcode)
    {
        case LIGHT_LIGHTNESS_OPCODE_STATUS:
            index = 0;
            break;
        case LIGHT_LIGHTNESS_LINEAR_OPCODE_STATUS:
```

```
            index = 1;
            break;
        case LIGHT_LIGHTNESS_LAST_OPCODE_STATUS:
            index = 2;
            break;
        case LIGHT_LIGHTNESS_DEFAULT_OPCODE_STATUS:
            index = 3;
            break;
        case LIGHT_LIGHTNESS_RANGE_OPCODE_STATUS:
            index = 4;
            break;
    }

static const mesh_opcode_handler_t m_opcode_handlers[] =
{
    {LIGHT_LIGHTNESS_OPCODE_STATUS, actual_linear_status_handle},
    {LIGHT_LIGHTNESS_LINEAR_OPCODE_STATUS, actual_linear_status_handle},
    {LIGHT_LIGHTNESS_LAST_OPCODE_STATUS, last_status_handle},
    {LIGHT_LIGHTNESS_DEFAULT_OPCODE_STATUS, dft_status_handle},
    {LIGHT_LIGHTNESS_RANGE_OPCODE_STATUS, range_status_handle},
};
static void actual_linear_status_handle(const mesh_model_msg_ind_t *p_rx_msg,  void *p_args)
{
    light_ln_client_t * p_client = (light_ln_client_t *) p_args;
    light_ln_status_params_t in_data = {0};

    if (p_rx_msg->msg_len == LIGHT_LIGHTNESS_STATUS_MINLEN || p_rx_msg->msg_len ==
 LIGHT_LIGHTNESS_STATUS_MAXLEN)
    {
        light_ln_status_msg_pkt_t * p_msg_params_packed = (light_ln_status_msg_pkt_t *)
 p_rx_msg->msg;

        if (p_rx_msg->msg_len == LIGHT_LIGHTNESS_STATUS_MINLEN)
        {
            in_data.present_ln = p_msg_params_packed->present_ln;
            in_data.target_ln = p_msg_params_packed->present_ln;
            in_data.remaining_time_ms = 0;
        }
        else
        {
            in_data.present_ln = p_msg_params_packed->present_ln;
            in_data.target_ln = p_msg_params_packed->target_ln;
            in_data.remaining_time_ms = model_transition_time_decode(p_msg_params_packed-
>remaining_time);
        }
        if(p_rx_msg->opcode.company_opcode == LIGHT_LIGHTNESS_OPCODE_STATUS)
        {
            p_client->settings.p_callbacks->ln_actual_status_cb(p_client, p_rx_msg,
 &in_data);
        }
        else
        {
            p_client->settings.p_callbacks->ln_linear_status_cb(p_client, p_rx_msg,
 &in_data);
        }

    }
}
static void last_status_handle(const mesh_model_msg_ind_t *p_rx_msg,  void *p_args)
{
    light_ln_client_t * p_client = (light_ln_client_t *) p_args;
    light_ln_last_status_params_t in_data = {0};
```

```
    if (p_rx_msg->msg_len == LIGHT_LIGHTNESS_LAST_STATUS_LEN)
    {
        light_ln_last_status_msg_pkt_t * p_msg_params_packed =
 (light_ln_last_status_msg_pkt_t *) p_rx_msg->msg;
        in_data.ln = p_msg_params_packed->ln;
        p_client->settings.p_callbacks->ln_last_status_cb(p_client, p_rx_msg, &in_data);
    }
}

static void dft_status_handle(const mesh_model_msg_ind_t *p_rx_msg, void *p_args)
{
    …
    if (p_rx_msg->msg_len == LIGHT_LIGHTNESS_DFT_STATUS_LEN)
    {
        light_ln_dft_status_msg_pkt_t * p_msg_params_packed =
                                        (light_ln_dft_status_msg_pkt_t *) p_rx_msg->msg;
        in_data.ln = p_msg_params_packed->ln;
        p_client->settings.p_callbacks->ln_dft_status_cb(p_client,
                                                         p_rx_msg,
                                                         &in_data);
    }
}
static void range_status_handle(const mesh_model_msg_ind_t *p_rx_msg,  void *p_args)
{
    light_ln_client_t * p_client = (light_ln_client_t *) p_args;
    light_ln_range_status_params_t in_data = {0};

    if (p_rx_msg->msg_len == LIGHT_LIGHTNESS_RANGE_STATUS_LEN)
    {
        light_ln_range_status_msg_pkt_t * p_msg_params_packed =
 (light_ln_range_status_msg_pkt_t *) p_rx_msg->msg;
        in_data.max_ln = p_msg_params_packed->max_ln;
        in_data.min_ln = p_msg_params_packed->min_ln;
        in_data.status_code = p_msg_params_packed->status_code;
        p_client->settings.p_callbacks->ln_range_status_cb(p_client, p_rx_msg, &in_data);
    }
}
```

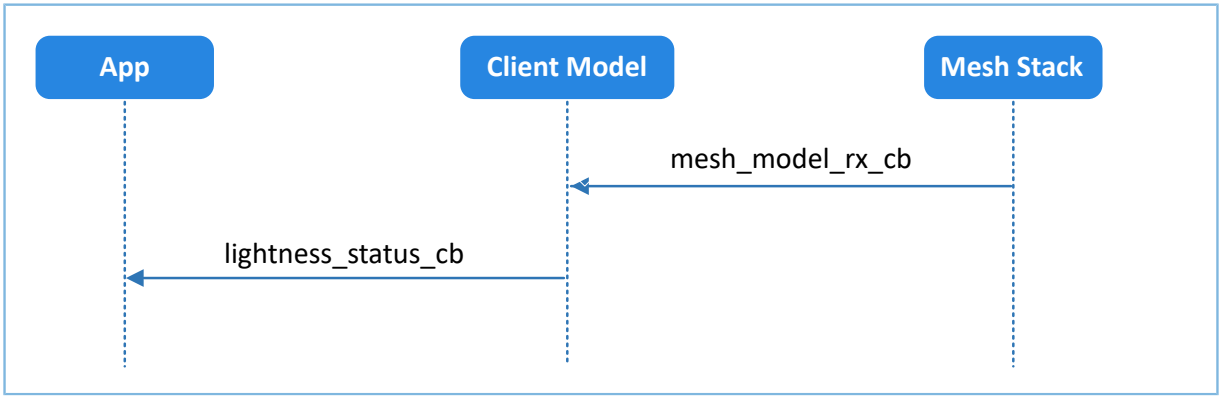

Figure 4-1 LIGHT_LIGHTNESS_OPCODE_STATUS message processing

4.2.1.1.3 Processing of To-be-sent Messages

The to-be-sent messages of the LLN Client Model are divided into three types: GET, SET, and SET_UNACK messages.

- LIGHT_LIGHTNESS_OPCODE_SET message

  To send this message, the second parameter of mesh_model_publish() shall carry reliable_info, which contains opcode and wait time of the massage to be replied.

```
uint16_t light_ln_client_set(light_ln_client_t * p_client,
                             const light_ln_set_params_t * p_params,
                             const model_transition_t * p_transition,
                             const cmd_type_t cmd_type)
{
    light_ln_set_msg_pkt_t set;
    bool reliable_trans_state = false;
    uint8_t tx_hdl = LIGHT_LIGHTNESS_CLIENT_SET_SEND_TX_HDL + p_client->model_instance_index
 * LIGHT_LIGHTNESS_CLIENT_TX_HDL_TOTAL;
    mesh_model_send_info_t model_msg_send;
    mesh_model_reliable_info_t reliable_info =
    {
        .reply_opcode.company_opcode = (cmd_type == ACTURL) ?
         LIGHT_LIGHTNESS_OPCODE_STATUS : LIGHT_LIGHTNESS_LINEAR_OPCODE_STATUS,
        .reply_opcode.company_id= MESH_ACCESS_COMPANY_ID_NONE,
        .status_cb = p_client->settings.p_callbacks->ack_transaction_status_cb,
        .timeout_ms = p_client->settings.timeout_ms,
    };
    ……
    if (MESH_ERROR_NO_ERROR == mesh_model_reliable_trans_is_on(
                                                            p_client->model_lid,
                                                            &reliable_trans_state))
    {
        if (reliable_trans_state)
        {
            return MESH_ERROR_SDK_RELIABLE_TRANS_ON;
        }
        else
        {
            uint8_t msg_length = message_set_packet_create(&set,
                                                           p_params,
                                                           p_transition);
            message_create(&model_msg_send, p_client->model_lid,
                           (cmd_type == ACTURL) ? LIGHT_LIGHTNESS_OPCODE_SET :
                            LIGHT_LIGHTNESS_LINEAR_OPCODE_SET,
                            tx_hdl, (uint8_t *)&set, msg_length);

            return mesh_model_publish(&model_msg_send, &reliable_info);
        }
    }
    else
    {
        return MESH_ERROR_SDK_INVALID_PARAM;
    }
}
```

After the SET message is sent, the LLN Client Model notifies users of the operation state using a callback function, as shown below.

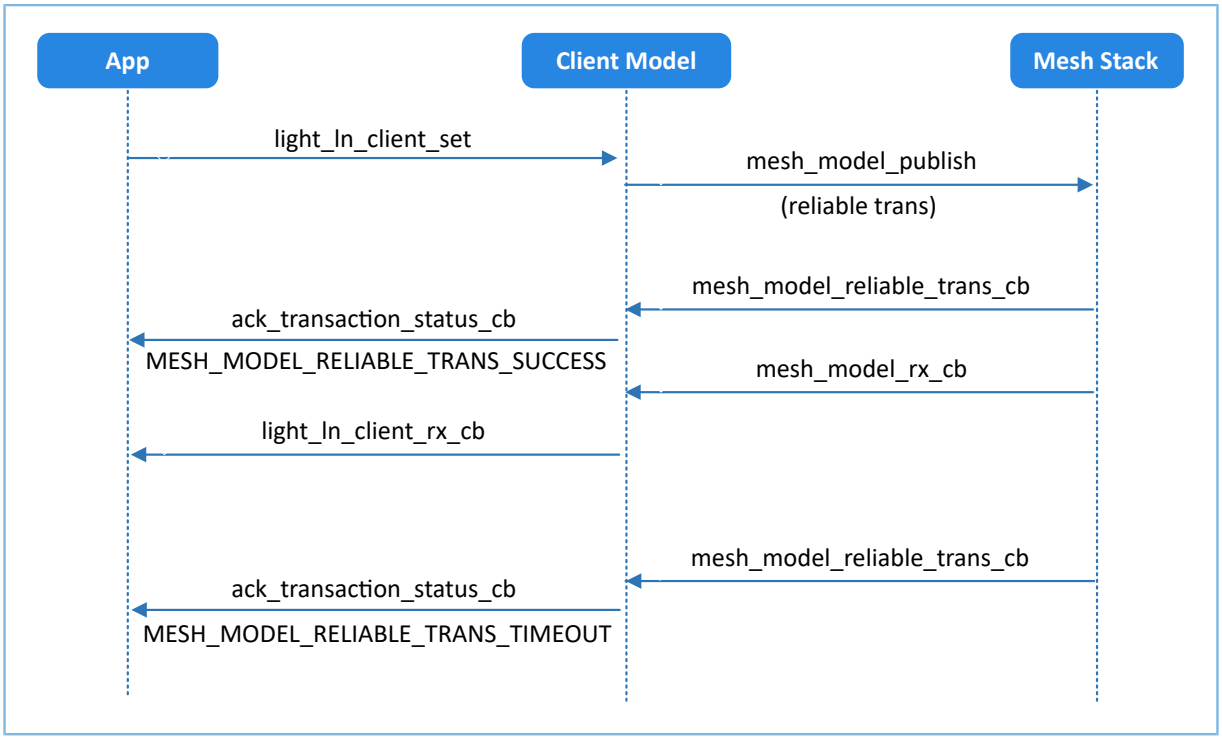

Figure 4-2 LIGHT_LIGHTNESS_OPCODE_SET message sending

- LIGHT_LIGHTNESS_OPCODE_SET_UNACKNOWLEDGED message

  To send this message, the second parameter of mesh_model_publish() shall be set to NULL. If no response from the server model is required, the LLN Client Model can send this message.

```
uint16_t light_ln_client_set_unack(light_ln_client_t * p_client,
                                   const light_ln_set_params_t * p_params,
                                   const model_transition_t * p_transition,
                                   const cmd_type_t cmd_type)
{
    light_ln_set_msg_pkt_t set_un;
    uint8_t tx_hdl = LIGHT_LIGHTNESS_CLIENT_SET_UNRELIABLE_SEND_TX_HDL +
             p_client->model_instance_index * LIGHT_LIGHTNESS_CLIENT_TX_HDL_TOTAL;
    mesh_model_send_info_t model_msg_send;
    …

    uint8_t msg_length = message_set_packet_create(&set_un,
                                                   p_params, p_transition);

    message_create(&model_msg_send, p_client->model_lid,
             (cmd_type == ACTURL) ? LIGHT_LIGHTNESS_OPCODE_SET_UNACKNOWLEDGED :
                                    LIGHT_LIGHTNESS_LINEAR_OPCODE_SET_UNACKNOWLEDGED,
                                        tx_hdl, (uint8_t *)&set_un, msg_length);

    return mesh_model_publish(&model_msg_send, NULL);
}
```

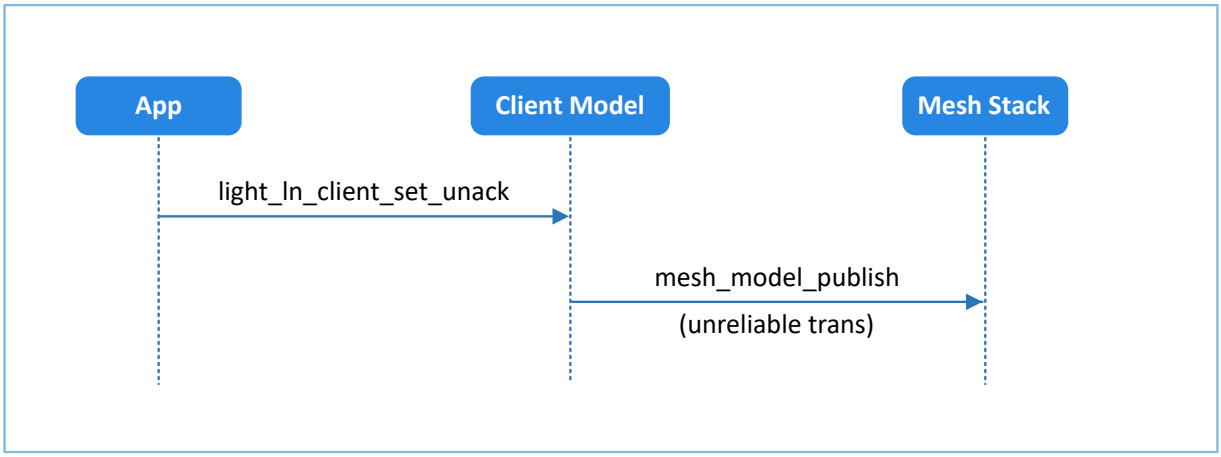

Figure 4-3 LIGHT_LIGHTNESS_OPCODE_SET_UNACKNOWLEDGED message sending

- LIGHT_LIGHTNESS_OPCODE_GET message

  To send this message, the second parameter of mesh_model_publish() shall carry reliable_info, which contains opcode and wait time of the massage to be replied.

```
uint16_t light_ln_client_get(light_ln_client_t * p_client,
                             const cmd_type_t cmd_type)
{
    mesh_model_send_info_t model_msg_send;
    bool reliable_trans_state = false;
    uint8_t tx_hdl = LIGHT_LIGHTNESS_CLIENT_GET_SEND_TX_HDL +
              p_client->model_instance_index * LIGHT_LIGHTNESS_CLIENT_TX_HDL_TOTAL;
    mesh_model_reliable_info_t reliable_info =
    {
        .reply_opcode.company_opcode =
                            (cmd_type == ACTURL) ? LIGHT_LIGHTNESS_OPCODE_STATUS :
                                                   LIGHT_LIGHTNESS_LINEAR_OPCODE_STATUS,
        .reply_opcode.company_id= MESH_ACCESS_COMPANY_ID_NONE,
        .status_cb = p_client->settings.p_callbacks->ack_transaction_status_cb,
        .timeout_ms = p_client->settings.timeout_ms,
    };

    …

    if (MESH_ERROR_NO_ERROR ==
                            mesh_model_reliable_trans_is_on(p_client->model_lid,
                                                            &reliable_trans_state))
    {
        if (reliable_trans_state)
        {
            return MESH_ERROR_SDK_RELIABLE_TRANS_ON;
        }
        else
        {
            message_create(&model_msg_send, p_client->model_lid,
                           (cmd_type == ACTURL) ? LIGHT_LIGHTNESS_OPCODE_GET :
                                                  LIGHT_LIGHTNESS_LINEAR_OPCODE_GET,
                                                  tx_hdl, NULL, 0);

            return mesh_model_publish(&model_msg_send, &reliable_info);
        }
    }
```

```
    else
    {
        return MESH_ERROR_SDK_INVALID_PARAM;
    }
}
```

After the GET message is sent, the LLN Client Model notifies users of the operation state using a callback function, as shown below.

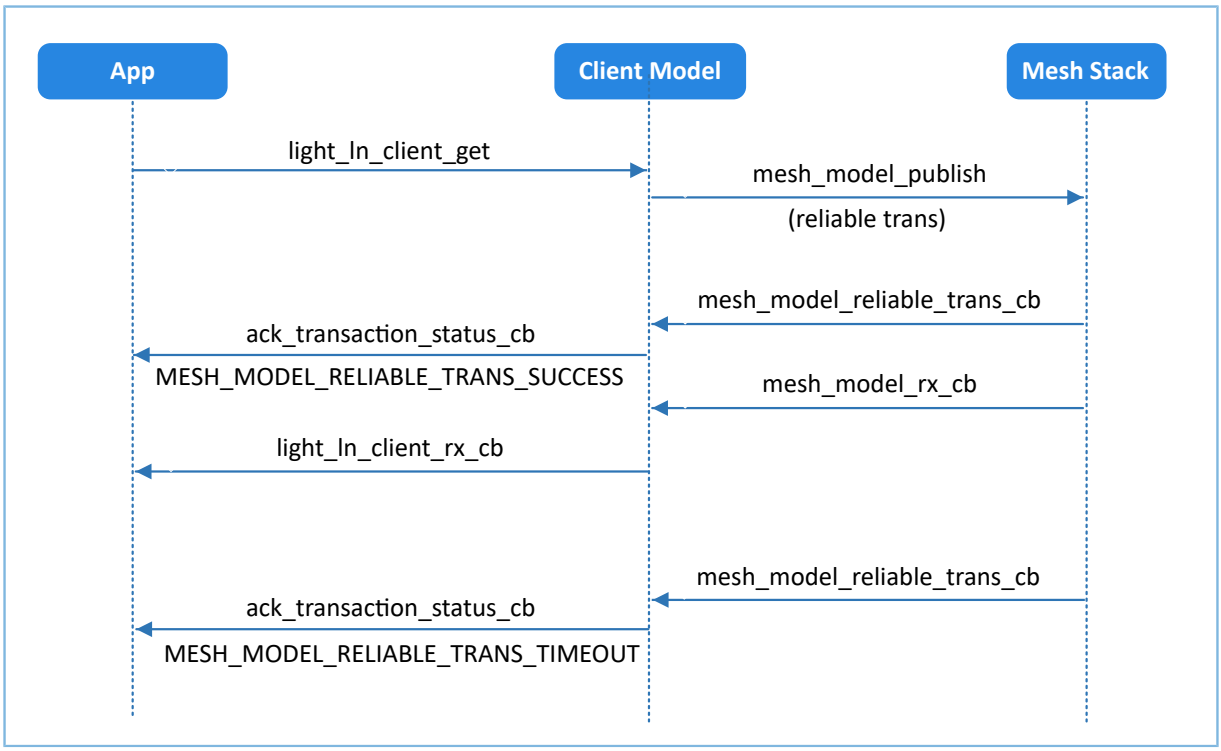

Figure 4-4 LIGHT_LIGHTNESS_OPCODE_GET message sending

4.2.1.2 LLN Server Model

The example code is in SDK_Folder\components\mesh\models\SIG\Lighting\light_lightness\src\light_lightness_server.c.

4.2.1.2.1 Processable Message List

The Mesh messages to be received and processed by the LLN Server Model are listed in the processable message list light_lightness_server_opcode_list.

```
static const uint16_t light_lightness_server_opcode_list[] =
{
    LIGHT_LIGHTNESS_OPCODE_GET,
    LIGHT_LIGHTNESS_OPCODE_SET ,
    LIGHT_LIGHTNESS_OPCODE_SET_UNACKNOWLEDGED,
    LIGHT_LIGHTNESS_LINEAR_OPCODE_GET,
    LIGHT_LIGHTNESS_LINEAR_OPCODE_SET,
    LIGHT_LIGHTNESS_LINEAR_OPCODE_SET_UNACKNOWLEDGED,
    LIGHT_LIGHTNESS_LAST_OPCODE_GET,
    LIGHT_LIGHTNESS_DEFAULT_OPCODE_GET,
    LIGHT_LIGHTNESS_RANGE_OPCODE_GET,
```

```
};
```

4.2.1.2.2 Callback Function Handling List

The callback function list light_lightness_server_msg_cb consists of three parts:

- cb_rx: message callback function received by the LLN Server Model
- cb_sent: callback function published or returned by the LLN Server Model to indicate message state
- cb_publish_period: callback function when the publish period state of the LLN Server Model changes

```
static const mesh_model_cb_t light_lightness_server_msg_cb =
{
    .cb_rx              = light_lightness_server_rx_cb,
    .cb_sent            = light_lightness_server_sent_cb ,
    .cb_publish_period  = NULL ,
};
```

The code snippet for processing of Mesh messages received by light_lightness_server_rx_cb() is as follows.

```
static void light_lightness_server_rx_cb(mesh_model_msg_ind_t *p_model_msg,
                                         void *p_args)
{
    uint16_t company_opcode = p_model_msg->opcode.company_opcode;

    mesh_opcode_handler_cb_t handler = NULL;

    for(uint8_t i = 0; i<light_lightness_server_register_info.num_opcodes; i++)
    {
        if(company_opcode == m_opcode_handlers[i].opcode)
        {
            handler = m_opcode_handlers[i].handler;
            break;
        }
    }

    if (NULL ! = handler)
    {
        handler(p_model_msg, p_args);
    }
}

static const mesh_opcode_handler_t m_opcode_handlers[] =
{
    {LIGHT_LIGHTNESS_OPCODE_GET, handle_get_cb},
    {LIGHT_LIGHTNESS_OPCODE_SET, handle_set_cb},
    {LIGHT_LIGHTNESS_OPCODE_SET_UNACKNOWLEDGED, handle_set_cb},
    {LIGHT_LIGHTNESS_LINEAR_OPCODE_GET, handle_get_cb},
    {LIGHT_LIGHTNESS_LINEAR_OPCODE_SET, handle_set_cb},
    {LIGHT_LIGHTNESS_LINEAR_OPCODE_SET_UNACKNOWLEDGED, handle_set_cb},
    {LIGHT_LIGHTNESS_LAST_OPCODE_GET, handle_get_cb},
    {LIGHT_LIGHTNESS_DEFAULT_OPCODE_GET, handle_get_cb},
    {LIGHT_LIGHTNESS_RANGE_OPCODE_GET, handle_get_cb},
};
```

- handle_set_cb() can process the following messages:
    - LIGHT_LIGHTNESS_OPCODE_SET

- LIGHT_LIGHTNESS_OPCODE_SET_UNACKNOWLEDGED
- LIGHT_LIGHTNESS_LINEAR_OPCODE_SET
- LIGHT_LIGHTNESS_LINEAR_OPCODE_SET_UNACKNOWLEDGED

The following introduces code implementation of handle_set_cb() by taking LIGHT_LIGHTNESS_OPCODE_SET message processing as an example.

```
static void handle_set_cb(const mesh_model_msg_ind_t *p_rx_msg,  void *p_args)
{
    light_ln_server_t  * p_server = (light_ln_server_t  *) p_args;
    APP_LOG_INFO("SERVER[%d] -- Receive message, want to set light lightness state!!!" ,
 p_server->model_instance_index);

    if (set_params_validate(p_rx_msg))
    {
        light_ln_state_set_cb_t set_cb_local = NULL;
        light_ln_set_params_t in_data = {0};
        model_transition_t in_data_tr = {0};
        light_ln_status_params_u out_data = {0};
        light_ln_set_msg_pkt_t * p_msg_params_packed = (light_ln_set_msg_pkt_t
                                                        *) p_rx_msg->msg;
        bool ack_flag = false;

        in_data.ln = gx_read16p ((void const *)&p_msg_params_packed->ln);
        in_data.tid = p_msg_params_packed->tid;

        if (model_tid_validate(&p_server->tid_tracker, p_rx_msg,
            p_rx_msg->opcode.company_opcode, in_data.tid))
        {
            ……

            switch(p_rx_msg->opcode.company_opcode)
            {
                case LIGHT_LIGHTNESS_OPCODE_SET:
                    ack_flag = true;
                    ......

                    set_cb_local = p_server->settings.p_callbacks->light_ln_cbs.set_cb;
                    break;
                ……

                default:
                    break;
            }

            if (NULL ! = set_cb_local)
            {
                set_cb_local(p_server,
                             p_rx_msg,
                             &in_data,
                             ((p_rx_msg->msg_len == LIGHT_LIGHTNESS_SET_MINLEN)
                             &&(p_server->p_dtt_ms == NULL)) ? NULL : &in_data_tr,
                             (ack_flag) ? &out_data : NULL);
            }

            if (ack_flag)
            {
                (void) status_send(p_server, p_rx_msg, &out_data);
```

```
            }
        }
    }
}

static uint32_t status_send(light_ln_server_t * p_server,
                            const mesh_model_msg_ind_t *p_rx_msg,
                            const light_ln_status_params_u * p_params)
{
    light_ln_status_msg_pkt_t *msg_pkt = NULL;
    light_ln_last_status_msg_pkt_t *msg_pkt_l = NULL;
    light_ln_dft_status_msg_pkt_t *msg_pkt_d = NULL;
    light_ln_range_status_msg_pkt_t *msg_pkt_r = NULL;
    uint8_t *p_send_data = NULL;
    uint16_t send_op = LIGHT_LIGHTNESS_OPCODE_STATUS;
    uint16_t data_send_len = 0;
    mesh_error_t status = MESH_ERROR_SDK_INVALID_PARAM;
    uint8_t tx_hdl = (NULL == p_rx_msg) ? LIGHT_LIGHTNESS_SERVER_PUBLISH_SEND_TX_HDL +
 p_server->model_instance_index * LIGHT_LIGHTNESS_SERVER_TX_HDL_TOTAL
                                        : LIGHT_LIGHTNESS_SERVER_RSP_SEND_TX_HDL + p_server-
>model_instance_index * LIGHT_LIGHTNESS_SERVER_TX_HDL_TOTAL;


    if (NULL ! = p_rx_msg)
    {
        switch(p_rx_msg->opcode.company_opcode)
        {
        ……

            case LIGHT_LIGHTNESS_OPCODE_SET:
                msg_pkt = (light_ln_status_msg_pkt_t *)sys_malloc(
                                                    sizeof(light_ln_status_msg_pkt_t));
                p_send_data = (uint8_t *)msg_pkt;
                ……

                gx_write16p( (void const *)&(msg_pkt->present_ln),
                             p_params->ln.present_ln);
                data_send_len += 2;
                ……
            break;
            ……

            default:
            break;
        }
}
    ……

    mesh_model_send_info_t msg_send =
    {
        .model_lid = p_server->model_lid,
        .opcode = MESH_ACCESS_OPCODE_SIG(send_op),
        .tx_hdl = tx_hdl,
        .p_data_send = (uint8_t *) p_send_data,
        .data_send_len = data_send_len,
        .dst = (NULL == p_rx_msg) ? MESH_INVALID_ADDR : p_rx_msg->src,
        .appkey_index =
                (NULL == p_rx_msg) ? MESH_INVALID_KEY_INDEX : p_rx_msg->appkey_index,
    };

    if (NULL == p_rx_msg)
    {
```

```
        APP_LOG_INFO("SERVER[%d] -- to publish : send opcode = %04X,
                     data_send_len = %d." ,
                     p_server->model_instance_index, send_op, data_send_len);
        status = mesh_model_publish(&msg_send, NULL);
    }
    else
    {
        APP_LOG_INFO("SERVER[%d] -- recv opcode = %04X, response opcode = %04X,
                     data_send_len = %d." ,
                     p_server->model_instance_index,
                     p_rx_msg->opcode.company_opcode, send_op, data_send_len);
        status = mesh_model_rsp_send(&msg_send);
    }

    if(p_send_data)
    {
        sys_free(p_send_data);
        p_send_data = NULL;
    }

    return status;
}
```

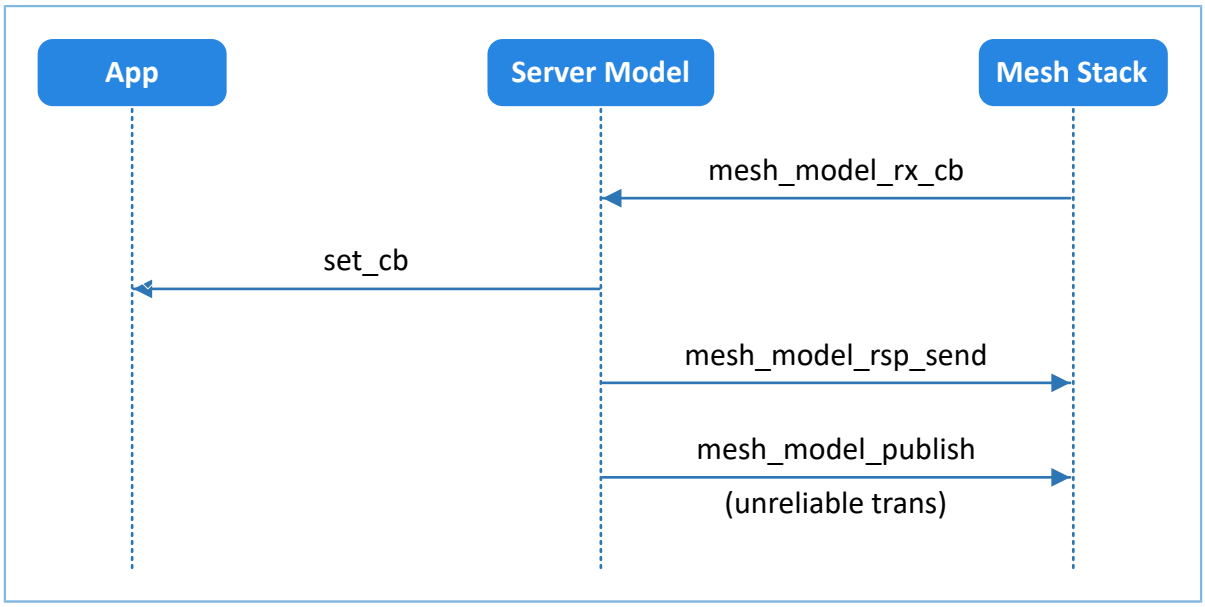

Figure 4-5 LIGHT_LIGHTNESS_OPCODE_SET message processing

The *_SET and *_SET_UNACKNOWLEDGE messages are processed in different ways. The former needs to call status_send() one more time than the latter, to call mesh_model_rsp_send() to respond to messages. Both the two message types finally call status_send() to call mesh_model_publish() to publish messages.

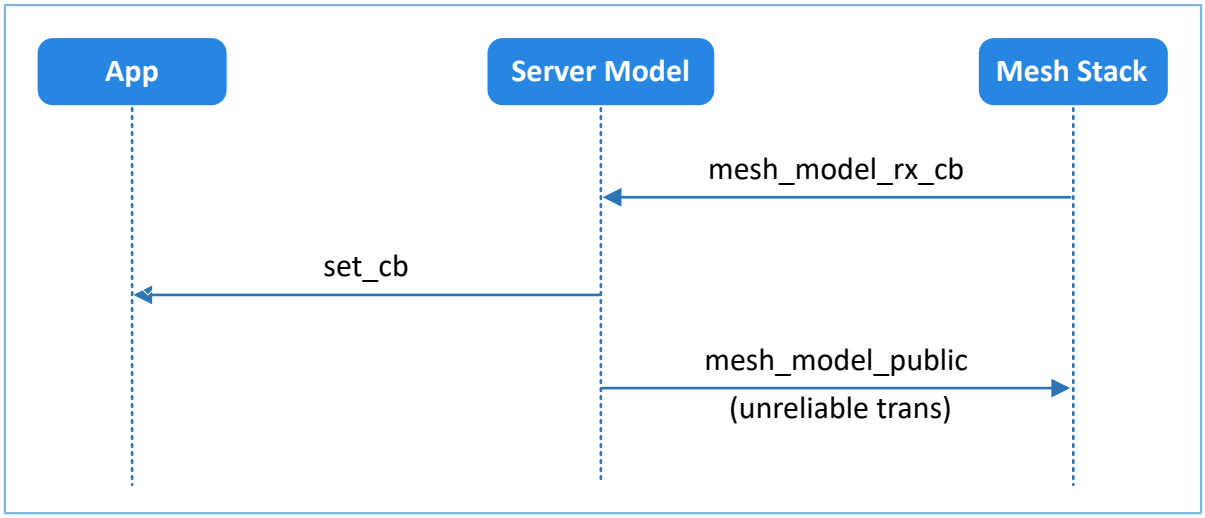

Figure 4-6 LIGHT_LIGHTNESS_OPCODE_SET_UNACKNOWLEDGED message processing

- The handle_get_cb() can process the following messages:
  - LIGHT_LIGHTNESS_OPCODE_GET
  - LIGHT_LIGHTNESS_LINEAR_OPCODE_GET
  - LIGHT_LIGHTNESS_LAST_OPCODE_GET
  - LIGHT_LIGHTNESS_DEFAULT_OPCODE_GET
  - LIGHT_LIGHTNESS_RANGE_OPCODE_GET

  status_send() calls mesh_model_rsp_send() to respond to messages.

```
static void handle_get_cb(const mesh_model_msg_ind_t *p_rx_msg,  void *p_args)
{
    light_ln_server_t * p_server = (light_ln_server_t *) p_args;
    light_ln_status_params_u out_data = {0};
    light_ln_state_get_cb_t get_cb_local = NULL;

    APP_LOG_INFO("[%s] enter." , __func__);
    APP_LOG_INFO("SERVER[%d] -- Receive message,
                  want to get state opcode %04X!!!" ,
                  p_server->model_instance_index,
                  p_rx_msg->opcode.company_opcode);

    if (get_params_validate(p_rx_msg))
    {
        switch (p_rx_msg->opcode.company_opcode)
        {
            case LIGHT_LIGHTNESS_OPCODE_GET:
                get_cb_local =
                            p_server->settings.p_callbacks->light_ln_cbs.get_cb;
                break;

            case LIGHT_LIGHTNESS_LINEAR_OPCODE_GET:
                get_cb_local =
                            p_server->settings.p_callbacks->light_ln_linear_cbs.get_cb;
                break;
```

```
            case LIGHT_LIGHTNESS_LAST_OPCODE_GET:
                get_cb_local =
                             p_server->settings.p_callbacks->light_ln_last_cbs.get_cb;
                break;

            case LIGHT_LIGHTNESS_DEFAULT_OPCODE_GET:
                get_cb_local =
                            p_server->settings.p_callbacks->light_ln_dft_cbs.get_cb;
                break;

            case LIGHT_LIGHTNESS_RANGE_OPCODE_GET:
                get_cb_local =
                            p_server->settings.p_callbacks->light_ln_range_cbs.get_cb;
                break;

            default:
                break;
        }

        if(get_cb_local)
        {
            get_cb_local(p_server, p_rx_msg, &out_data);
           (void) status_send(p_server, p_rx_msg, &out_data);
        }
    }
}
```

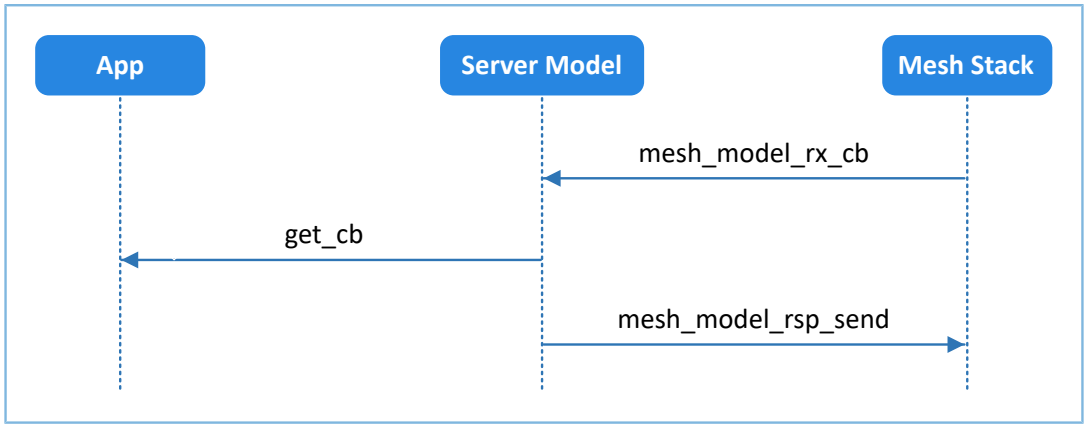

Figure 4-7 LIGHT_LIGHTNESS_OPCODE_GET message processing

4.2.1.2.3 Processing of To-be-sent Messages

The LLN Server Model can send the following messages actively:

- LIGHT_LIGHTNESS_LINEAR_OPCODE_STATUS
- LIGHT_LIGHTNESS_OPCODE_STATUS
- LIGHT_LIGHTNESS_LAST_OPCODE_STATUS
- LIGHT_LIGHTNESS_DEFAULT_OPCODE_STATUS
- LIGHT_LIGHTNESS_RANGE_OPCODE_STATUS

status_send() calls mesh_model_publish() to publish messages.

```
uint16_t light_ln_server_status_publish(light_ln_server_t * p_server,
                                        const light_ln_status_params_t * p_params)
{
    if (NULL == p_server || NULL == p_params)
    {
        return MESH_ERROR_SDK_INVALID_PARAM;
    }

    return status_send(p_server, NULL,
                       (const light_ln_status_params_u *) p_params);
}
```

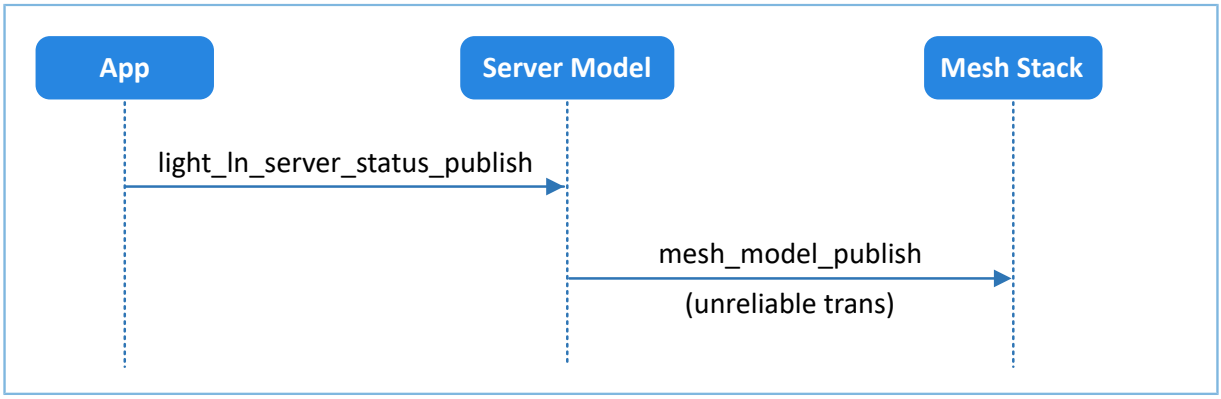

Figure 4-8 LIGHT_LIGHTNESS_OPCODE_STATUS message sending

4.2.1.3 LLNS Server Model

The example code is in SDK_Folder\components\mesh\models\SIG\Lighting\light_lightness\src\light_lightness_setup_server.c.

4.2.1.3.1 Processable Message List

The Mesh messages to be received and processed by the LLNS Server Model are listed in the processable message list light_lightness_setup_server_opcode_list.

```
static const uint16_t light_lightness_setup_server_opcode_list[] =
{
    LIGHT_LIGHTNESS_DEFAULT_OPCODE_SET,
    LIGHT_LIGHTNESS_DEFAULT_OPCODE_SET_UNACKNOWLEDGED,
    LIGHT_LIGHTNESS_RANGE_OPCODE_SET,
    LIGHT_LIGHTNESS_RANGE_OPCODE_SET_UNACKNOWLEDGED,
};
```

4.2.1.3.2 Callback Function Handling List

The LLNS Server Model does not support publishing messages. When the state changes, messages shall be published by the LLN Server Model.

The callback function list light_lightness_setup_server_msg_cb consists of three parts:

- cb_rx: message callback function received by the LLNS Server Model from the LLN Client

- cb_sent: callback function published or returned by the LLNS Server Model to indicate message state. If no message is published, set cb_sent to NULL.
- cb_publish_period: callback function when the publish period state of the LLNS Server Model changes. If no message is published, set cb_publish_period to NULL.

```
static const mesh_model_cb_t light_lightness_setup_server_msg_cb =
{
    .cb_rx = light_lightness_setup_server_rx_cb,
    .cb_sent = NULL,
    .cb_publish_period = NULL
};
```

4.2.1.3.3 Processing of To-be-sent Messages

After receiving and processing a range state set or default state set message, the LLNS Server Model does not respond to the message directly; instead, it responds to the message through the LLN Server Model.

This mechanism is implemented by setting the **model_lid** to be sent to **LLN Server Model**.

```
mesh_model_send_info_t msg_send =
{
  .model_lid = p_server->ln_server->model_lid,
  .opcode = MESH_ACCESS_OPCODE_SIG(send_op),
  .tx_hdl = tx_hdl,
  .p_data_send = (uint8_t *) p_send_data,
  .data_send_len = data_send_len,
  .dst = (NULL == p_rx_msg) ? MESH_INVALID_ADDR : p_rx_msg->src,
  .appkey_index =
          (NULL == p_rx_msg) ? MESH_INVALID_KEY_INDEX : p_rx_msg->appkey_index,
};
```

Code implementation of status_send() of the LLNS Server Model is described as follows:

```
static uint32_t status_send(light_ln_setup_server_t * p_server,
                                const mesh_model_msg_ind_t *p_rx_msg,
                                const light_ln_status_params_u * p_params)
{
    light_ln_dft_status_msg_pkt_t *msg_pkt_d = NULL;
    light_ln_range_status_msg_pkt_t *msg_pkt_r = NULL;
    uint8_t *p_send_data = NULL;
    uint16_t send_op = LIGHT_LIGHTNESS_DEFAULT_OPCODE_STATUS;
    uint16_t data_send_len = 0;
    mesh_error_t status = MESH_ERROR_SDK_INVALID_PARAM;
    uint8_t tx_hdl = LIGHT_LIGHTNESS_SERVER_RSP_SEND_TX_HDL + p_server->model_instance_index
 * LIGHT_LIGHTNESS_SERVER_TX_HDL_TOTAL;

    if (NULL == p_rx_msg)
    {
        return MESH_ERROR_SDK_INVALID_PARAM;
    }
    else
    {
        switch(p_rx_msg->opcode.company_opcode)
        {
            case LIGHT_LIGHTNESS_DEFAULT_OPCODE_SET:
                send_op = LIGHT_LIGHTNESS_DEFAULT_OPCODE_STATUS;
                msg_pkt_d = (light_ln_dft_status_msg_pkt_t
 *)sys_malloc(sizeof(light_ln_dft_status_msg_pkt_t));
```

```
                p_send_data = (uint8_t *)msg_pkt_d;
                if(NULL == p_send_data)
                {
                    return MESH_ERROR_INSUFFICIENT_RESOURCES;
                }

                gx_write16p( (void const *)&(msg_pkt_d->ln), p_params->ln_dft.ln);
                data_send_len += 2;
            break;

            case LIGHT_LIGHTNESS_RANGE_OPCODE_SET:

                if (p_params->ln_range.status_code ! = STATUS_CODES_SUCCESS)
                {
                    return MESH_ERROR_NO_ERROR;
                }
                send_op = LIGHT_LIGHTNESS_RANGE_OPCODE_STATUS;
                msg_pkt_r = (light_ln_range_status_msg_pkt_t
 *)sys_malloc(sizeof(light_ln_range_status_msg_pkt_t));
                p_send_data = (uint8_t *)msg_pkt_r;
                if(NULL == p_send_data)
                {
                    return MESH_ERROR_INSUFFICIENT_RESOURCES;
                }

                msg_pkt_r->status_code = p_params->ln_range.status_code;
                data_send_len += 1;
                gx_write16p( (void const *)&(msg_pkt_r->min_ln), p_params->ln_range.min_ln);
                data_send_len += 2;
                gx_write16p( (void const *)&(msg_pkt_r->max_ln), p_params->ln_range.max_ln);
                data_send_len += 2;
            break;

            default:
            break;
        }

        if ((0 == data_send_len) || (NULL == p_send_data))
        {
            return MESH_ERROR_INSUFFICIENT_RESOURCES;
        }

        if (p_server->ln_server)//setup server send status
        {
            mesh_model_send_info_t msg_send =
            {
                .model_lid = p_server->ln_server->model_lid,
                .opcode = MESH_ACCESS_OPCODE_SIG(send_op),
                .tx_hdl = tx_hdl,
                .p_data_send = (uint8_t *) p_send_data,
                .data_send_len = data_send_len,
                .dst = (NULL == p_rx_msg) ? MESH_INVALID_ADDR : p_rx_msg->src,
                .appkey_index = (NULL == p_rx_msg) ? MESH_INVALID_KEY_INDEX : p_rx_msg-
>appkey_index,
            };

            APP_LOG_INFO("SERVER[%d] -- to response , recv opcode:%04X, response opcode:
%04X, data length:%d." ,
                    p_server->model_instance_index, p_rx_msg->opcode.company_opcode,
 send_op, data_send_len);
            status = mesh_model_rsp_send(&msg_send);
        }
```

```
        else
        {
            APP_LOG_ERROR("[ERROR] -- No server to send status");
            status = MESH_ERROR_COMMAND_DISALLOWED;
        }

        if(p_send_data)
        {
            sys_free(p_send_data);
            p_send_data = NULL;
        }

        return status;
    }
}
```

4.2.2 State Processing of LLN Models

This section demonstrates how to implement state processing for standard models by introducing the implementation code for state processing of LLN Models.

4.2.2.1 State Binding of LLN Models

The lightness state is bound with generic level state and generic onoff state respectively. Users can refer to the following code and make some modifications as needed if state binding is required when users customize a model.

1. Initialize the binding list.

   Before registering models in SDK_Folder\projects\mesh\SIG\mesh_app_light_lightness_server\Src\user\user_app.c, call mesh_model_bind_list_init() to initialize the binding list for future use during subsequent state registration and binding.

```
#define USER_MODEL_REG_NUM_MAX 0x20

static void mesh_init_cmp_callback(void)
{
    //set run time
    mesh_run_time_set(1000, 0);
    //init bind model list
    mesh_model_bind_list_init(USER_MODEL_REG_NUM_MAX);
    //register model
    if (MESH_ERROR_NO_ERROR == app_model_server_init())
    {
        APP_LOG_INFO("Register model successful!" );
        mesh_enable();
    }
}
```

2. Bind models and register callback functions.

   During server model initialization in SDK_Folder\projects\mesh\SIG\common\src\app_light_lightness_server.c, call mesh_model_bind_list_add() to register the model binding information to the binding list. If any other model state is to be bound with the lightness state, the registered app_light_ln_bind_cb() will be called to transmit data for binding.

```
#define MODEL_ID_LIGHTS_LN          (0x1300)
```



```
……
uint16_t app_light_ln_server_init(app_light_lightness_server_t *p_server,
                                  uint8_t element_offset,
                                  app_light_ln_set_cb_t set_cb,
                                  app_light_ln_get_cb_t get_cb)
{
    …
mesh_model_bind_list_add(MODEL_ID_LIGHTS_LN,
                         element_offset, (void *)p_server,
                         app_light_ln_bind_cb);
…
}
```

3. Process binding information.

   In SDK_Folder\projects\mesh\SIG\common\src\app_light_lightness_server.c, the registered app_light_ln_bind_cb() implements binding state changes based on transmitted information such as **model id** and **state**. Below is the implementation code for binding the generic onoff state after the Generic OnOff Server Model calls app_onoff_bind_check().

```
#define MODEL_ID_GENS_OO           (0x1000)

static bool app_light_ln_bind_cb(void *p_server, uint32_t src_model_id,
                                 void *p_data, uint16_t data_len)
{
    app_light_lightness_server_t * pl_server =
                                      (app_light_lightness_server_t *)p_server;
    bool ret = false;

    APP_LOG_INFO("[%s] enter, SERVER[%d], src model %04X data length %d",
                 __func__, pl_server->server.model_instance_index,
                 src_model_id, data_len);

    if (pl_server ! = NULL)
    {
        switch(src_model_id)
        {
            …
            case MODEL_ID_GENS_OO:
            {
                if (data_len == 1)
                {
                    uint8_t onoff = *(uint8_t *)p_data;

                    if ((onoff == 0x00) && (pl_server->state.state.present_ln ! =
                                                              0))//onoff :off
                    {
                        pl_server->state.state.present_ln = 0x00;
                        pl_server->state.state.present_ln_linear = 0x00;
                        lightness_state_value_update(pl_server, true);

                        ret = true;
                    }
                    else if ((onoff == 0x01) &&
                             (pl_server->state.state.present_ln == 0)) //onoff:on
                    {
                        if(pl_server->state.state.default_ln == 0x0000)
                        {
                            pl_server->state.state.present_ln =
                                                  pl_server->state.state.last_ln;
```

```
                        }
                        else
                        {
                            pl_server->state.state.present_ln =
                                                pl_server->state.state.default_ln;
                        }
                        pl_server->state.state.present_ln_linear =
                               CV_LIGHTS_LN_LINEAR(pl_server->state.state.present_ln);
                        lightness_state_value_update(pl_server, true);

                        ret = true;
                    }
                    APP_LOG_INFO("model gen onoff bind message length %d,
                                  value %d, lightness actual %d", data_len, onoff,
                                  pl_server->state.state.present_ln);
                }
                break;
            }
            …
            default:
                break;
        }

        if(ret)
        {
            app_light_ln_status_publish(pl_server);
            app_light_ln_bind_check(pl_server, src_model_id);
        }
    }

    return ret;
}
```

4. Check the binding state.

   The LLN Server Model needs to process binding messages from other server models (such as Generic OnOff Server Model) during running. In addition, when the lightness state changes, the LLN Server Model needs to send the new state to other server models to allow them to process the binding relationship. For example, if the LLN Server Model notifies the Generic OnOff Server Model of binding the generic onoff state, call mesh_model_bind_check() to implement the binding operation.

```
static void lightness_state_value_update(
                        app_light_lightness_server_t * p_server, bool immed)
{
    …
      if (!p_server->value_updated)
      {
          p_server->light_ln_set_cb(p_server->server.model_instance_index,
                                    &(p_server->state.state));
          p_server->value_updated = true;
          app_light_ln_bind_check(p_server,MODEL_ID_LIGHTS_LN);
      }
  …
}

static void app_light_ln_bind_check(app_light_lightness_server_t *p_server,
                                    uint32_t trigger_model)
{
    …
```

```
    if (trigger_model ! = MODEL_ID_GENS_OO)
    {
        /* check bind generic onoff state */
        mesh_model_bind_check(p_server->server.model_instance_index,
                              MODEL_ID_GENS_OO, MODEL_ID_LIGHTS_LN,
                              &p_server->state.state.present_ln,
                              sizeof(p_server->state.state.present_ln));
    }
…
    return ;
}
```

5 FAQ

This chapter describes possible problems, reasons, and solutions during running of the LLN Models examples.

5.1 Why Does Project Download Fail?

- Description

  Project download with Keil fails.

- Analysis

  Keil is not configured correctly for project download.

- Solution

  Make sure Keil is configured correctly, as shown below.

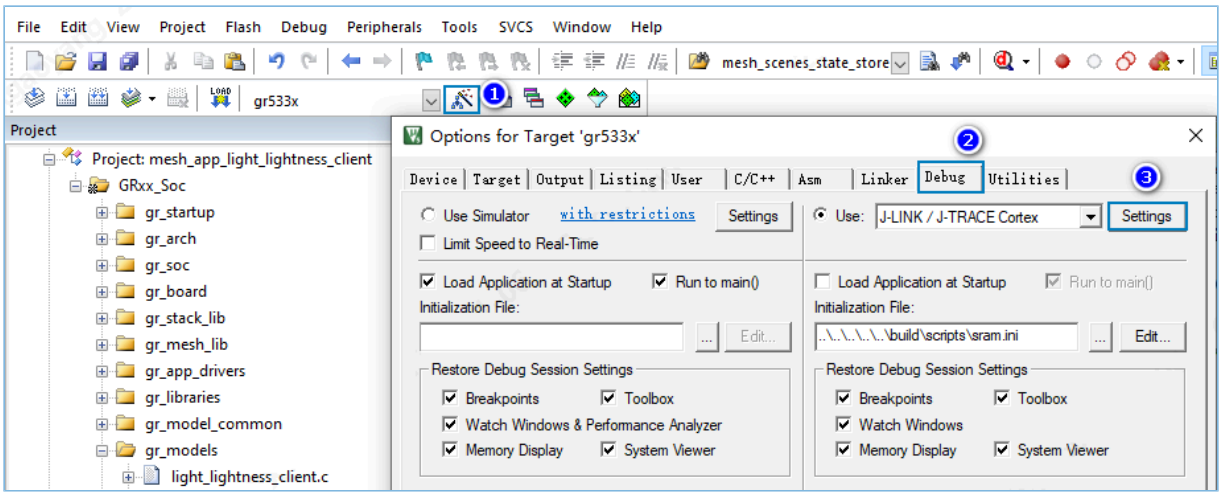

Figure 5-1 Keil download configuration

5.2 Why Does Device Initialization Fail?

- Description

  After the target file is downloaded and runs normally, no **Enable successful** is printed on GRUart, indicating device initialization fails.

- Analysis

  Model registration for the configured project fails.

- Solution

  - The registered lightness model number in the configured project cannot exceed the number defined in the macro LIGHT_LIGHTNESS_INSTANCE_COUNT.
  - The number defined in the macro LIGHT_LIGHTNESS_INSTANCE_COUNT cannot exceed the number defined in LIGHT_LIGHTNESS_SERVER_INSTANCE_COUNT_MAX.

- The number of all registered models in the configured project cannot exceed the number defined in the macro MESH_MODEL_NUM_MAX.

5.3 Why Does GRMesh Fail to Discover Any Device?

- Description

  GRMesh (Android) fails to discover any device for networking.

- Analysis

  The firmware runs abnormally, or the previous networking information is not cleared.

- Solution

  - Check whether the device name in the project has been changed; make sure the device name is shorter than 16 bytes.
  - Erase all information in the GR533x SoC, and then re-download firmware.
  - Check whether **enable successful** is printed on GRUart.

5.4 Why Does Data Transmission/Reception Between LLN Client and LLN Server Fail?

- Description

  Data transmission/reception between LLN Client and LLN Server fails after successful networking.

- Analysis

  The firmware is not correctly downloaded to the board, or the BaudRate on the GRUart is incorrect, resulting in failure to print logs.

- Solution

  - Make sure the publish address corresponds to the subscription address between the networked LLN Client and LLN Server.
  - Make sure the App keys of LLN Client and LLN Server are consistent.

5.5 Why Does Binding Triggering Fail?

- Description

  Model state changes fail to trigger corresponding changes in the binding state.

- Analysis

  The parameters of the function called during binding registration or binding state check are incorrect.

  - Solution

    - Make sure model_instance_index() is called correctly during binding registration.

- Make sure the src model id, dest model id, and model index are input correctly during binding state check.
- Make sure the transmitted data and data length are correct during binding state check.
- Make sure the transmitted messages corresponding to the model id are processed in the callback handler.