



Bluetooth Mesh Developer Study Guide

Bluetooth Mesh - Hands-on Coding Lab - Dimmer Node

Version: 1.0.0

Last updated: 15th June 2018

Contents

REVISION HISTORY	4
EXERCISE 4 – IMPLEMENTING THE DIMMER	5
Introduction	5
The Generic Level Client Model	5
Project Set Up	5
Addresses	6
Security Keys	6
Explanation	7
Key Indices and IV Index	7
Miscellaneous Variables	7
Explanation	7
Configuration Client and Server Models	7
Explanation	7
Explanation	8
Health Server Model	8
Explanation	8
Generic Level Message Types	8
Explanation	8
RX Messages and Handler Functions	8
Explanation	8
Explanation	9
Node Composition	9
Explanation	9
Explanation	10
Checkpoint	10
Device UUID	10
Bluetooth Stack Initialisation	10
Explanation	11
Bluetooth Mesh Initialisation	11
Checkpoint	12
Self-Provisioning	12
Explanation	12
Self-Configuration	12

Explanation	13
Checkpoint	14
RX Messages	14
Generic Level Status	15
Explanation	16
Checkpoint	16
TX Messages - Absolute Level Changes	16
Generic Level Get	17
Explanation	17
The Generic Level Set message types	17
Explanation	18
Checkpoint	18
Generic Level Set Unacknowledged	18
Generic Level Set	19
Checkpoint	19
TX Messages - Relative Level Changes	19
Product Design Requirements	20
The Generic Delta Set message types	20
Explanation	21
Checkpoint	21
Generic Delta Set	21
Generic Delta Set Unacknowledged	22
Checkpoint	23
TX Messages - Move Level Operations	23
The Generic Move Set message types	23
Explanation	24
Checkpoint	25
Generic Move Set	25
Generic Move Set Unacknowledged	26
Checkpoint	27
Next	27

Revision History

Version	Date	Author	Changes
1.0.0	15 th June 2018	Martin Woolley Bluetooth SIG	Initial version.
1.0.2	16 th August 2018	Martin Woolley Bluetooth SIG	Fixed description of move set operations and added test for sending a generic move set with delta level = 0 to stop a running move on the target light node(s).

Exercise 4 – Implementing the Dimmer

Introduction

Your next coding exercise will involve implementing the code required to turn one of your devices into a Bluetooth mesh dimmer control. You'll test against the same micro:bit which acted as a light when you tested your on/off switch node in the previous exercise.

Many of the initial steps of this exercise will be the same or very similar to corresponding steps in Exercise 3. Seeing and going through them again will be good experience and help you see the general pattern that Zephyr mesh applications tend to conform to.

The Generic Level Client Model

The dimmer node will implement the Generic Level Client Model. This model defines messages which collectively allow you to set the generic level state of target devices (Generic Level Server Models) to an absolute level value, to a relative level known as a *delta* or to initiate a dynamic transition from one state to another, known as a *move*. Generic Level Servers can report level state values by sending a Generic Level Status message.






Project Set Up

Create the following directories for your project:

```
dimmer/  
  build/  
  src/
```

Copy all of the files and the src directory from \$MDG\code\start_state\Dimmer\src to your project's root directory.

Your project directory should contain the following files:

Name	Date modified	Type	Size
 src	01/05/2018 14:12	File folder	
 CMakeLists.txt	15/02/2018 15:11	Text Document	1 KB
 prj.conf	15/02/2018 15:16	CONF File	1 KB
 prj_bbc_microbit.conf	30/04/2018 13:53	CONF File	1 KB
 sample.yaml	01/02/2018 13:03	YAML File	1 KB

The src folder should contain a single file, main.c which contains only skeleton code. You'll complete the implementation per the requirements of the dimmer node in this exercise.

Prepare your project by executing the following commands:

```
cd build  
cmake -GNinja -DBOARD=bbc_microbit ..
```

Your starter code should compile and link. Check that this is the case by executing the build command *ninja* from your build directory.

Now open main.c in your editor and proceed with the exercise.

Addresses

All nodes must have a unique unicode address. Our dimmer will send messages addressed to a group to which one or more lights will subscribe. Add the following code under the *addresses* comment to define these items:

```
#define NODE_ADDR 0x0002
#define GROUP_ADDR 0xc000
static u16_t target = GROUP_ADDR;
static u16_t node_addr = NODE_ADDR;
```

Security Keys

To allow *self-provisioning* to be used, we must choose and hard-code three key values. Add the following code under the comment *security keys*:

```
// 0123456789abcdef0123456789abcdef
static const u8_t dev_key[16] = {
    0x01,
    0x23,
    0x45,
    0x67,
    0x89,
    0xab,
    0xcd,
    0xef,
    0x01,
    0x23,
    0x45,
    0x67,
    0x89,
    0xab,
    0xcd,
    0xef,
};

// 0123456789abcdef0123456789abcdef
static const u8_t net_key[16] = {
    0x01,
    0x23,
    0x45,
    0x67,
    0x89,
    0xab,
    0xcd,
    0xef,
    0x01,
    0x23,
    0x45,
    0x67,
    0x89,
    0xab,
    0xcd,
    0xef,
};

// 0123456789abcdef0123456789abcdef
static const u8_t app_key[16] = {
    0x01,
    0x23,
    0x45,
    0x67,
    0x89,
    0xab,
    0xcd,
    0xef,
    0x01,
    0x23,
    0x45,
    0x67,
    0x89,
```

```
        0xab,  
        0xcd,  
        0xef,  
    };
```

Explanation

We've defined three 128-bit key values; the *DevKey* which is used to secure application layer communication when configuring the device, *AppKey*, which secures application layer in other message exchanges and *NetKey* which secures the lower layers of the communication protocol. We'll specify these keys as parameters to various function calls.

Key Indices and IV Index

Define the following constants to act as our NetKey and AppKey key indices and IV Index value:

```
// 4.3.1.1 Key indexes  
static const u16_t net_idx;  
static const u16_t app_idx;  
// 3.8.4 IV Index  
static const u32_t iv_index;
```

Miscellaneous Variables

Define the following variables under the *other* comment.

```
static u8_t flags;  
static u8_t tid;
```

Explanation

tid is the transaction identifier field which transactional messages require.

flags is a Zephyr API field used when provisioning.

Configuration Client and Server Models

A node must support the Configuration Server Model. We're self-configuring so our node must also be able to act as a configuration client and it also needs the Configuration Client model. Add the following code under the *Configuration Client* comment:

```
static struct bt_mesh_cfg_cli cfg_cli = {};
```

Explanation

This struct is all we need to be able to indicate that this model is part of our node's composition. We'll come on to the topic of Node Composition shortly.

Add the following code under the *Configuration Server* comment:

```
static struct bt_mesh_cfg_srv cfg_srv = {  
    .relay = BT_MESH_RELAY_DISABLED,  
    .beacon = BT_MESH_BEACON_DISABLED,  
    .frnd = BT_MESH_FRIEND_NOT_SUPPORTED,  
    .gatt_proxy = BT_MESH_GATT_PROXY_NOT_SUPPORTED,  
    .default_ttl = 7,  
    /* 3 transmissions with 20ms interval */  
    .net_transmit = BT_MESH_TRANSMIT(2, 20),  
};
```

Explanation

The `bt_mesh_cfg_srv` struct contains state values with which to initialise our configuration server model, which we'll do later. As you can see, the network roles of *relay*, *beacon*, *friend* and *proxy* have been disabled since we do not need them. We've also set a default TTL of 7 and indicated that each network PDU must be retransmitted twice, at intervals of 20ms. This is to increase the reliability of our network.

Health Server Model

A node must also support the Health Server Model, which is concerned with node diagnostics. Add the following code under the *Health Server* comment:

```
BT_MESH_HEALTH_PUB_DEFINE(health_pub, 0);
static struct bt_mesh_health_srv health_srv = {};
```

Explanation

This model can publish diagnostics messages and so we start by using the Zephyr SDK `BT_MESH_HEALTH_PUB_DEFINE` macro to define a publication context. We then define a context for the model with a struct of type `bt_mesh_health_srv`. We'll use each of these items when we declare the models our node supports.

Generic Level Message Types

Under the *operations supported by this model* comment, add the following message opcode definitions:

```
#define BT_MESH_MODEL_OP_GEN_LEVEL_GET      BT_MESH_MODEL_OP_2(0x82, 0x05)
#define BT_MESH_MODEL_OP_GEN_LEVEL_SET      BT_MESH_MODEL_OP_2(0x82, 0x06)
#define BT_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK BT_MESH_MODEL_OP_2(0x82, 0x07)
#define BT_MESH_MODEL_OP_GEN_LEVEL_STATUS   BT_MESH_MODEL_OP_2(0x82, 0x08)
#define BT_MESH_MODEL_OP_GEN_DELTA_SET      BT_MESH_MODEL_OP_2(0x82, 0x09)
#define BT_MESH_MODEL_OP_GEN_DELTA_SET_UNACK BT_MESH_MODEL_OP_2(0x82, 0x0A)
#define BT_MESH_MODEL_OP_GEN_MOVE_SET       BT_MESH_MODEL_OP_2(0x82, 0x0B)
#define BT_MESH_MODEL_OP_GEN_MOVE_SET_UNACK BT_MESH_MODEL_OP_2(0x82, 0x0C)
```

Explanation

We've defined constants for each of the message types that are part of the generic level client model so that we can reference them more easily elsewhere in our code.

RX Messages and Handler Functions

We need to specify the message opcodes which each model is required to be able to receive and process and for each message opcode, a function which will handle messages of that type.

Add this code after the `#define` constants you added in the previous step.

```
static const struct bt_mesh_model_op gen_level_cli_op[] = {
    {BT_MESH_MODEL_OP_GEN_LEVEL_STATUS, 1, generic_level_status},
    BT_MESH_MODEL_OP_END,
};
```

Explanation

This array of `bt_mesh_model_op` types contains a single significant item, which specifies the opcode for the Generic Level Status message, the only type of message which our Generic Level Client must

be able to receive and process. It specifies that access message payloads must be at least 1 octet long and a function called `generic_level_status` which will handle all such messages received.

`BT_MESH_MODEL_END` indicates the end of the definition.

Let's add a skeleton definition of the `generic_level_status` function now.

Under the comment *handler functions for this model's RX messages* add the following:

```
static void generic_level_status(struct bt_mesh_model *model,
                                struct bt_mesh_msg_ctx *ctx,
                                struct net_buf_simple *buf)
{
    printk("generic_level_status\n");
}
```

Explanation

generic level status messages received by this device will cause a call to the `generic_level_status` function because we registered it as a handler for messages with that opcode in a `bt_mesh_model_op` struct above.

The handler function currently logs a message to any attached serial terminal with *printk*.

Node Composition

A mesh node consists of one or more elements, each of which contains one or more models. This hierarchical arrangement is called the *node composition* and is something which would normally be set by an external configuration client application. We're using *self configuration* and so our code will act as both configuration client and configuration server and the composition state definition will be hard coded.

Find the section with the comment heading "Composition". Add the following code under the comment:

```
static struct bt_mesh_model sig_models[] = {
    BT_MESH_MODEL_CFG_SRV(&cfg_srv),
    BT_MESH_MODEL_CFG_CLI(&cfg_cli),
    BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
    BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_LEVEL_CLI, gen_level_cli_op,
                  NULL, 0),
};
```

Explanation

We've created an array of model definitions using the Zephyr SDK `bt_mesh_model` type and using macros from the SDK which make it easy to define special models like the configuration client and server models and the health server model. We've used a general purpose model definition macro to define the generic level client model. As you can see, these definitions reference the definitions we prepared earlier on.

We've defined our models, so now we need to define the element(s) which contain them and the node which contains the element(s). Add the following code in the *Composition* section under the `sig_models` definition.

```
// node contains elements. Note that BT_MESH_MODEL_NONE means "none of this type" and here means "no vendor models"
static struct bt_mesh_elem elements[] = {
```

```

        BT_MESH_ELEM(0, sig_models, BT_MESH_MODEL_NONE),
    };

    // node
    static const struct bt_mesh_comp comp = {
        .elem = elements,
        .elem_count = ARRAY_SIZE(elements),
    };

```

Explanation

We've used the Zephyr SDK's `BT_ELEM_MACRO` to define an element and indicated that it contains the models we defined in the `sig_models` array. We've also defined a struct called *comp* (for 'composition') which effectively acts as the top of a hierarchical definition of the node and its composition, starting with the elements directly owned by the node.

Checkpoint

Build your code with the *ninja* command from within your *build/* directory. It should compile and link now but will generate a number of warnings relating to items we've defined but not yet used.

If you get any errors, check your code against the fragments specified in this document. If necessary, check the full solution in the `code\solution\Dimmer` directory.

Device UUID

Before a device has been provisioned and configured, it does not have its unique unicast mesh address and nor does it have a DevKey. To allow unique identification of an unprovisioned device during the provisioning process, all devices must have a unique UUID called the Device UUID. The Device UUID is set by the manufacturer during the manufacturing process.

Add the following under the *device UUID* comment. Note that this value is different to the one we assigned to the Switch node in the previous exercise.

```

// cfa0ea7e-17d9-11e8-86d1-5f1ce28adea2
static const uint8_t dev_uuid[16] = { 0xcf, 0xa0, 0xea, 0x7e, 0x17, 0xd9, 0x11, 0xe8, 0x86,
0xd1, 0x5f, 0x1c, 0xe2, 0x8a, 0xde, 0xa2};

```

Bluetooth Stack Initialisation

Your next job is to initialise the Bluetooth and Bluetooth mesh stacks. After that we'll self-provision and self-configure.

Update your main function as shown:

```

void main(void)
{
    int err;
    printk("dimmer\n");
    configureButtons();

    err = bt_enable(bt_ready);
    if (err)
    {
        printk("bt_enable failed with err %d\n", err);
    }
}

```

Explanation

Our main function calls a function *configureButtons()* which is already present in the starter code. This just sets up GPIO so that pressing buttons A or B on the micro:bit results in a callback to an associated handler function. We'll come to those functions and decide what we want to happen when the buttons are pressed soon.

bt_enable is a Zephyr API function which enables Bluetooth. On completion, the system makes a callback to the function provided as an argument, in our case, the function *bt_ready*.

Bluetooth Mesh Initialisation

The Zephyr API defines a type, *bt_mesh_prov* which allows a struct containing various provisioning related properties to be defined. We'll need this for our initialisation of the mesh stack. Add the following under the *provisioning properties and capabilities* comment:

```
static const struct bt_mesh_prov prov = {
    .uuid = dev_uuid,
};
```

Add the *bt_ready* function above *main* with the following code:

```
static void bt_ready(int err)
{
    if (err)
    {
        printk("bt_enable init failed with err %d\n", err);
        return;
    }
    printk("Bluetooth initialised OK\n");
    err = bt_mesh_init(&prov, &comp);
    if (err)
    {
        printk("bt_mesh_init failed with err %d\n", err);
        return;
    }
    printk("Mesh initialised OK\n");
}
```

Checkpoint

Build your code with the *ninja* command from within your *build/* directory. It should compile and link now but will generate a number of warnings relating to items we've defined but not yet used.

If you get any errors, check your code against the fragments specified in this document. If necessary, check the full solution in the `code\solution\Switch` directory.

Self-Provisioning

Update the *bt_ready* function to call a new function, *selfProvision* which you'll add shortly.

```
static void bt_ready(int err)
{
    if (err)
    {
        printk("bt_enable init failed with err %d\n", err);
        return;
    }
    printk("Bluetooth initialised OK\n");
    err = bt_mesh_init(&prov, &comp);
    if (err)
    {
        printk("bt_mesh_init failed with err %d\n", err);
        return;
    }
    printk("Mesh initialised OK\n");
    err = selfProvision();
}
```

Add the *selfProvision* function above the *bt_ready* function:

```
static int selfProvision(void)
{
    // now we provision ourselves... this is not how it would normally be done!
    int err = bt_mesh_provision(net_key, net_idx, flags, iv_index, node_addr, dev_key);
    if (err)
    {
        printk("Provisioning failed (err %d)\n", err);
        return err;
    }
    printk("Provisioning completed\n");

    return 0;
}
```

Explanation

Using the Zephyr APIs to self-provision just entails calling the *bt_mesh_provision* function with the arguments as shown. As you can see, this includes providing the device with its NetKey, unicast node address and DevKey.

Self-Configuration

Update *bt_ready* with a call to a new function, *selfConfigure*.

```
static void bt_ready(int err)
{
    if (err)
    {
        printk("bt_enable init failed with err %d\n", err);
        return;
    }
    printk("Bluetooth initialised OK\n");
    err = bt_mesh_init(&prov, &comp);
    if (err)
```

```

{
    printk("bt_mesh_init failed with err %d\n", err);
    return;
}
printk("Mesh initialised OK\n");
err = selfProvision();
selfConfigure();
printk("provisioned, configured and ready\n");
}

```

Now add the *selfConfigure* function above *selfProvision*:

```

static int selfConfigure(void)
{
    int err;
    tid = 0;
    printk("self-configuring...\n");

    /* Add Application Key */
    err = bt_mesh_cfg_app_key_add(net_idx, node_addr, net_idx, app_idx, app_key, NULL);
    if (err)
    {
        printk("ERROR adding appkey (err %d)\n", err);
        return err;
    }
    else
    {
        printk("added appkey\n");
    }

    /* Bind to Health model */
    err = bt_mesh_cfg_mod_app_bind(net_idx, node_addr, node_addr, app_idx,
BT_MESH_MODEL_ID_HEALTH_SRV, NULL);
    if (err)
    {
        printk("ERROR binding to health server model (err %d)\n", err);
        return err;
    }
    else
    {
        printk("bound appkey to health server model\n");
    }

    /* Bind to level client model */
    err = bt_mesh_cfg_mod_app_bind(net_idx, node_addr, node_addr, app_idx,
BT_MESH_MODEL_ID_GEN_LEVEL_CLI, NULL);
    if (err)
    {
        printk("ERROR binding to level client model (err %d)\n", err);
        return err;
    }
    else
    {
        printk("bound appkey to level client model\n");
    }

    printk("self-configuration complete\n");
    return 0;
}

```

Explanation

In this function, we're using Zephyr's configuration client model APIs to make changes to our configuration server model's states. As a reminder, this is just a trick to avoid needing to use a separate configuration application running on a smartphone. It's not something you would do when developing a commercial product.

We accomplish three things in this function.

1. Using `bt_mesh_cfg_app_key_add` we add our AppKey to the node and at the same time, bind it to the NetKey. AppKeys are always associated with or *bound to* a specific NetKey.
2. We then call `bt_mesh_cfg_mod_app_bind` to bind our AppKey to the generic level client model so that this key is used for securing the fields in generic level client messages, relating to the higher layers of the stack.
3. Finally, we then call `bt_mesh_cfg_mod_app_bind` to bind our AppKey to the health server model so that this key is used for securing the fields in health server messages, relating to the higher layers of the stack.

Checkpoint

Build your code with *ninja*. There will still be a few warnings produced, but most will now have been dealt with.

```
C:\workspaces\zephyr_projects\mdk_dimmer_solution\build>ninja
[1/100] Generating always_rebuild
Building for board bbc_microbit
[2/7] Building C object CMakeFiles/app.dir/src/main.c.obj
../src/main.c:210:14: warning: 'mapDimmerSettingToDelta' defined but not used [-Wunused-
function]
    static s32_t mapDimmerSettingToDelta(s8_t setting_change) {
        ^~~~~~
../src/main.c:28:14: warning: 'target' defined but not used [-Wunused-variable]
    static ul6_t target = GROUP_ADDR;
        ^~~~~~
[4/7] Linking C executable zephyr\zephyr_prebuilt.elf
Memory region      Used Size  Region Size  %age Used
    FLASH:         124188 B      256 KB       47.37%
     SRAM:          15552 B       16 KB       94.92%
    IDT_LIST:         132 B        2 KB        6.45%
[7/7] Linking C executable zephyr\zephyr.elf
```

RX Messages

Our dimmer implements the configuration client and server models, the health server model and the generic level client model. Here's an extract from the mesh model specification showing the rules regarding the generic level client and message support:

Element	SIG Model ID	Procedure	Messages	Rx	Tx
Main	0x1003	Generic Level	Generic Level Get		○
			Generic Level Set		○
			Generic Level Set Unacknowledged		○
			Generic Delta Set		○
			Generic Delta Set Unacknowledged		○
			Generic Move Set		○
			Generic Move Set Unacknowledged		○
			Generic Level Status	C.1	

C.1: If any of the messages: Generic Level Get, Generic Level Set, Generic Delta Set, Generic Move Set are supported, the Generic Level Status message shall also be supported; otherwise, support for the Generic Level Status message is optional.

Table 3.117: Generic Level Client elements and messages

Our dimmer node will support setting the target light nodes to a specified absolute level or up or down by a given delta value. It will also support initiating a dynamic transition known as a *move*. Therefore, we'll implement the Get, Set, Delta and Move messages as well as the Level Status message.

As you should recall from Exercise 3, we don't need to do anything regarding the messages supported by the configuration and health models since these are taken care of by the Zephyr framework.

The only RX message that the dimmer must support is the generic level status message. A status message is a type of mesh message which contains a state value, reported by a server model. Status messages are sent as responses to GET messages, acknowledged SET messages but can also be sent at any time by the server.

Generic Level Status

In the *RX Messages and Handler Functions* section of this exercise, we registered a function to handle generic level status messages and added a skeleton implementation of that function. Let's complete it now so that the micro:bit logs the value of the level state in the received status message to the serial console.

When implementing a handler function for any mesh message, your first stop should be the mesh model specification to look at the structure of the message and the rules governing the presence or absence of fields in the message. Here's the Generic Level Status message:

Field	Size (octets)	Notes
Present Level	2	The present value of the Generic Level state.
Target Level	2	The target value of the Generic Level state (Optional).
Remaining Time	1	Format as defined in Section 3.1.3 (C.1).

C.1: If the Target Level field is present, the Remaining Time field shall also be present; otherwise these fields shall not be present.

Table 3.44: Generic Level Status message parameters

The Present Level field identifies the present Generic Level state of the element (see Section 3.1.2).

If present, the Target Level field identifies the target Generic Level state that the element is to reach (see Section 3.1.2).

If present, the Remaining Time field identifies the time that it will take the element to complete the transition to the target Generic Level state of the element (see Section 3.1.2).

Update the `generic_level_status` function so that it looks like this:

```
static void generic_level_status(struct bt_mesh_model *model,
                                struct bt_mesh_msg_ctx *ctx,
                                struct net_buf_simple *buf)
{
    printk("generic_level_status\n");
    u8_t buflen = buf->len;
    s16_t present_level = (s16_t) net_buf_simple_pull_le16(buf);
```

```

printk("present_level=%d\n",present_level);
if (buflen > 3) {
    s16_t target_level = (s16_t) net_buf_simple_pull_le16(buf);
    printk("target_level=%d\n",target_level);
    if (buflen > 4) {
        u8_t remaining_time = net_buf_simple_pull_u8(buf);
        printk("remaining_time=%d\n",remaining_time);
    } else {
        printk("ERROR: remaining time field is missing\n");
    }
}
}
}

```

Explanation

Earlier in the exercise, you defined an array called `gen_level_cli_op` which maps message opcodes to functions, including in this case, the opcode for the generic level status message, which you associated with the `generic_level_status` function:

```

static const struct bt_mesh_model_op gen_level_cli_op[] = {
    {BT_MESH_MODEL_OP_GEN_LEVEL_STATUS, 1, generic_level_status},
    BT_MESH_MODEL_OP_END,
};

```

You also associated this opcode/function mapping with the generic level client model in an array of models supported by the node's element:

```

static struct bt_mesh_model sig_models[] = {
    BT_MESH_MODEL_CFG_SRV(&cfg_srv),
    BT_MESH_MODEL_CFG_CLI(&cfg_cli),
    BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
    BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_LEVEL_CLI, gen_level_cli_op,
                  NULL, 0),
};

```

This allows received messages with the generic level status opcode to be routed to the `generic_level_status` function for processing.

In the handler function, we use the one of the [Zephyr buffer processing APIs](#) `net_buf_simple_pull_le16` to extract the level state value which is a 16 bit number with little endian byte ordering. We then write this value to the console.

We treat the other two fields as optional and check the buffer length to determine whether or not the two optional fields *target_level* and *remaining_time* are present and if they are, extract them into variables which we then log.

Checkpoint

Build your code with the *ninja* command from within your *build/* directory. It should compile and link now but will generate a number of warnings relating to items we've defined but not yet used.

TX Messages - Absolute Level Changes

We'll now implement the TX messages relating to setting absolute level values. We'll trigger sending TX messages using the two buttons on the front of the micro:bit and will select which specific message types we want to send by commenting out the functions called in our button handlers, as we did when testing the switch node in Exercise 3.

Generic Level Get

Add the following function to the *TX message producer functions* section:

```
void generic_level_get()
{
    // 2 bytes for the opcode
    // 0 bytes parameters:
    // 4 additional bytes for the TransMIC

    NET_BUF_SIMPLE_DEFINE(msg, 2 + 0 + 4);
    struct bt_mesh_msg_ctx ctx = {
        .net_idx = net_idx,
        .app_idx = app_idx,
        .addr = target,
        .send_ttl = BT_MESH_TTL_DEFAULT,
    };
    bt_mesh_model_msg_init(&msg, BT_MESH_MODEL_OP_GEN_LEVEL_GET);
    if (bt_mesh_model_send(&sig_models[3], &ctx, &msg, NULL, NULL))
    {
        printk("Unable to send generic level get message\n");
    }
    printk("level get message sent\n");
}
```

Update the *buttonB_work_handler* function to call *generic_level_get()*:

```
void buttonB_work_handler(struct k_work *work)
{
    printk("Button B work handler\n");
    generic_level_get();
}
```

Explanation

When sending any mesh access message using the Zephyr APIs, we need to create and populate a buffer with the message opcode, any associated parameters and the transport message integrity code which all mesh access messages must have at their end. Zephyr provides the `NET_BUF_SIMPLE_DEFINE` macro to help make that easier.

A message context, containing index values with which to reference netkey and appkey values, the destination address for the message and a TTL value is required and the type *bt_mesh_msg_ctx* is available for that purpose.

bt_mesh_model_msg_init completes the initialisation of the message PDU and *bt_mesh_model_send* sends the message.

The Generic Level Set message types

The code required to send a *generic level set unacknowledged* message only differs from the code used when sending an acknowledged *generic level set* message in that the message opcode is different. Consequently we'll use a common function for both message types, with the required message opcode as a parameter. Add the following function in the section headed *Generic Level Client - TX message producer functions*:

```
void generic_level_set(s16_t level, u32_t opcode)
{
    // 2 bytes for the opcode
    // 3 bytes of parameters / payload : 16 bits level, 8 bits TID
    // 4 additional bytes for the TransMIC
    NET_BUF_SIMPLE_DEFINE(msg, 9);
    bt_mesh_model_msg_init(&msg, opcode);
    net_buf_simple_add_le16(&msg, level);
}
```

```

net_buf_simple_add_u8(&msg, tid);

struct bt_mesh_msg_ctx ctx = {
    .net_idx = net_idx,
    .app_idx = app_idx,
    .addr = target,
    .send_ttl = BT_MESH_TTL_DEFAULT,
};
tid++;
if (bt_mesh_model_send(&sig_models[3], &ctx, &msg, NULL, NULL))
{
    printk("Unable to send generic level set (ack/unack) message\n");
}
printk("level set (ack/unack) message %d sent\n", level);
}

```

Explanation

The function takes two parameters, the first of which is the level state value to send. The second is an opcode which will indicate which of *generic level set* or *generic level set unacknowledged* we wish to send.

Consulting the mesh model specification for the definition of the *generic level set* message type, we see it has this structure:

Field	Size (octets)	Notes
Level	2	The target value of the Generic Level state
TID	1	Transaction Identifier
Transition Time	1	Format as defined in Section 3.1.3. (Optional)
Delay	1	Message execution delay in 5 millisecond steps (C.1)

C.1: If the Transition Time field is present, the Delay field shall also be present; otherwise these fields shall not be present.

We don't intend to use transition times for now and so the last two parameters are not required in our case.

Once again we use *NET_BUF_SIMPLE_DEFINE* to create a network buffer. We then create a message context and complete the initialisation of the required PDU using Zephyr API functions. We add the level state and TID (Transaction Identifier) values, increment the TID in readiness for sending our next message and then send the message using the Zephyr *bt_mesh_model_send* function.

Checkpoint

We'll perform a couple of quick tests now to ensure we're on the right track.

Generic Level Set Unacknowledged

Update *buttonA_work_handler* as shown:

```

void buttonA_work_handler(struct k_work *work)
{
    printk("buttonA_work_handler\n");
    // set unack random
    s16_t level = get_random_level();
    generic_level_set(level, BT_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK);
}

```

Build and install the binary on your dimmer node. Reset both the dimmer and light nodes by pressing the reset button. Test by pressing button A to send a *generic level set unacknowledged* message with a random level value and then press button B to send a *generic level get* message to retrieve the level state value that the previous message set. Monitor the whole process using serial consoles attached to both the dimmer and light nodes over USB. In the console attached to the dimmer node you should see something like the following:

```
buttonA_work_handler
level set (ack/unack) message 5930 sent
buttonB_work_handler
level get message sent
generic_level_status
present_level=5930
```

In the console attached to the light node you should see something like:

```
generic_level_set_unack
generic_level_set_common
level_state=5930 level_inx=5
generic_level_status transition_in_progress=0
generic_level_status transition_in_progress=0
30028> level status message 5930 sent
```

The LED display of the light node will illuminate a number of LEDs in proportion to the level value set given the supported range of 0-32767 (our light does not support negative level values).

Generic Level Set

Update buttonA_work_handler as shown:

```
void buttonA_work_handler(struct k_work *work)
{
    printk("buttonA_work_handler\n");
    // set unack random
    // s16_t level = get_random_level();
    // generic_level_set(level,BT_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK);

    // set random
    s16_t level = get_random_level();
    generic_level_set(level,BT_MESH_MODEL_OP_GEN_LEVEL_SET);
}
```

Checkpoint

Build and install the binary on your dimmer node.

```
C:\dimmer\build>copy zephyr\zephyr.hex e:
1 file(s) copied.
```

Reset both the dimmer and light nodes by pressing the reset button. Test by pressing button A to send a *generic level set* message with a random level value. Monitor the whole process using serial consoles attached to both the dimmer and light nodes over USB. You should see a *generic level status* message being automatically returned as an acknowledgement to your *generic level set* message.

TX Messages - Relative Level Changes

We'll now implement the TX messages relating to setting relative level values. Specifically, this will involve the *generic delta set unacknowledged* and *generic delta set* messages. These message types have the following structure:

Field	Size (octets)	Notes
Delta Level	4	The Delta change of the Generic Level state
TID	1	Transaction Identifier
Transition Time	1	Format as defined in Section 3.1.3. (Optional)
Delay	1	Message execution delay in 5 milliseconds steps (C.1)

C.1: If the Transition Time field is present, the Delay field shall also be present; otherwise these fields shall not be present.

Table 3.40: Generic Delta Set message parameters

Our dimmer will not use the transition time and delay fields when sending *generic delta set* messages.

Product Design Requirements

For this part of the exercise, imagine our dimmer control has a button marked “-” on the right and a button marked “+” on the left. Each time the “-” button is pressed, lights under control will reduce their level by *an appropriate amount* until they reach their minimum level, at which point pressing “-” will have no further effect. Similarly, repeatedly pressing “+” will increase the level of nodes under control until their maximum is reached.

How many increments or decrements of level should be required by the user to change the level of lights under control through the full supported range of level values? This is something we'd have to think about, largely considering user experience requirements. In our case, we want our dimmer to work as though lights can be at one of ten discreet levels so that pressing “+” 10 times will increase the level of a light from minimum (off) to maximum, one step at a time.

Other nodes should not know anything about such implementation or design issues. Lights should not know anything about the fact that this particular dimmer conceptually divides the full range of levels into 10 points or bands. Best practice dictates that the dimmer node should send a delta level value which is calculated as a multiple of one tenth of the full level state 0-32767 range. This is what the function `mapDimmerSettingToDelta` does for us (it was already present in the starter code).

The Generic Delta Set message types

The code required to send a *generic delta set unacknowledged* message only differs from the code used when sending an acknowledged *generic delta set* message in that the message opcode is different. Consequently we'll use a common function for both message types, with the required message opcode as a parameter. Add the following function in the section headed *Generic Level Client - TX message producer functions*:

```
void generic_delta_set(s32_t delta_level, u32_t opcode)
{
    // 2 bytes for the opcode
    // 5 bytes of parameters / payload : first 32 bits is the delta level, remaining 8 bits
    // is for the TID. We don't use the optional transition time or delay fields.
    // 4 additional bytes for the TransMIC
    NET_BUF_SIMPLE_DEFINE(msg, 2 + 5 + 4);
```

```

struct bt_mesh_msg_ctx ctx = {
    .net_idx = net_idx,
    .app_idx = app_idx,
    .addr = target,
    .send_ttl = BT_MESH_TTL_DEFAULT,
};
bt_mesh_model_msg_init(&msg, opcode);
net_buf_simple_add_le32(&msg, delta_level);
net_buf_simple_add_u8(&msg, tid);
tid++;
if (bt_mesh_model_send(&sig_models[3], &ctx, &msg, NULL, NULL))
{
    printk("Unable to send generic delta set (ack/unack) message\n");
}
printk("delta set (ack/unack) message %d sent\n", delta_level);
}

```

Explanation

The function takes two parameters, the first of which is the delta level state value to send. The second is an opcode which will indicate which of *generic delta set* or *generic delta set unacknowledged* we wish to send.

Once again we use *NET_BUF_SIMPLE_DEFINE* to create a network buffer. We then create a message context and complete the initialisation of the required PDU using Zephyr API functions. We add the level state and TID (Transaction Identifier) values, increment the TID in readiness for sending our next message and then send the message using the Zephyr *bt_mesh_model_send* function.

Checkpoint

We'll perform a couple of quick tests now by implementing some calls to our *generic_delta_set* function with various parameters.

Generic Delta Set

Update *buttonA_work_handler* as shown:

```

void buttonA_work_handler(struct k_work *work)
{
    printk("buttonA_work_handler\n");
    // set unack random
    // s16_t level = get_random_level();
    // generic_level_set(level, BT_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK);

    // set random
    // s16_t level = get_random_level();
    // generic_level_set(level, BT_MESH_MODEL_OP_GEN_LEVEL_SET);

    // delta set positive
    generic_delta_set(mapDimmerSettingToDelta(1), BT_MESH_MODEL_OP_GEN_DELTA_SET);
}

```

When button A is pressed, it will send a *generic delta set* message with a delta level corresponding to an increase of 1 tenth of the level full range. That value is calculated by the *mapDimmerSettingsToDelta* function.

Modify *buttonB_handler* so that it sends a *generic delta set* message with a delta level value mapped from -1.

```

void buttonB_work_handler(struct k_work *work)
{
    printk("buttonB_work_handler\n");
}

```

```

// generic_level_get();

// delta set negative
generic_delta_set(mapDimmerSettingToDelta(-1),BT_MESH_MODEL_OP_GEN_DELTA_SET);
}

```

Build and install the binary on your dimmer node. Reset both the dimmer and light nodes by pressing the reset button. Test by pressing button A ten times to send a series of ten *generic delta set unacknowledged* messages which cause the receiving light node to increase its level through successive, discreet steps from completely off (no LEDs lit) to fully on (all LEDs lit). Now press button B ten times to reverse the process. Monitor the whole process using serial consoles attached to both the dimmer and light nodes over USB, remembering that we're sending acknowledged messages and therefore the dimmer node should receive status messages in response to each *generic delta set* message it sends. In the console attached to the dimmer node you should see something like the following:

```

buttonB_work_handler
setting_change=1 mapped to delta=3276
delta set (ack/unack) message 3276 sent

```

In the console attached to the light node you should see something like:

```

level_state=3276 level_inx=3
4832> sending a level status work item to 2
generic_level_status transition_in_progress=0
generic_level_status transition_in_progress=0
4860> level status message 3276 sent

```

The LED display of the light node will illuminate a number of LEDs in proportion to the level state value incremented with the delta level in the *generic delta set* message.

Note that occasionally you may see what looks like more than one mesh message being sent in response to a single button press. This is an issue with the debounce GPIO support in Zephyr. More than one mesh message is being sent because the Zephyr framework is acting as though you pressed the button more than once.

Generic Delta Set Unacknowledged

Update buttonA_work_handler as shown:

```

void buttonA_work_handler(struct k_work *work)
{
    printk("buttonA_work_handler\n");
    // set unack random
    // s16_t level = get_random_level();
    // generic_level_set(level,BT_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK);

    // set random
    // s16_t level = get_random_level();
    // generic_level_set(level,BT_MESH_MODEL_OP_GEN_LEVEL_SET);

    // delta set positive
    // generic_delta_set(mapDimmerSettingToDelta(1),BT_MESH_MODEL_OP_GEN_DELTA_SET);

    // delta set unack positive
    generic_delta_set(mapDimmerSettingToDelta(1),BT_MESH_MODEL_OP_GEN_DELTA_SET_UNACK);
}

```

And update `buttonB_work_handler` to send negative, unacknowledged generic delta set messages like this:

```
void buttonB_work_handler(struct k_work *work)
{
    printk("buttonB_work_handler\n");
    // generic_level_get();

    // delta set negative
    // generic_delta_set(mapDimmerSettingToDelta(-1), BT_MESH_MODEL_OP_GEN_DELTA_SET);

    // delta set unack negative
    generic_delta_set(mapDimmerSettingToDelta(-1), BT_MESH_MODEL_OP_GEN_DELTA_SET_UNACK);
}
```

Checkpoint

Build, install and test again, remembering to reset both the dimmer and light micro:bits to ensure SEQ numbers do not get repeated (see Troubleshooting section). The test results should be as for the acknowledged *generic delta set* messages except that status messages will not be returned.

TX Messages - Move Level Operations

Generic move set messages initiate a dynamic transition in a receiving model which will execute at a certain speed, optionally with its start delayed by a specified amount. Move operations can continue indefinitely, wrapping around and continuing when upper or lower limits are reached or can stop when either of these limits is reached. It is an implementation decision as to which combination of these available behaviours products exhibit.

Another generic move set message from a client with a delta level of zero indicates that the transition should stop.

Our light node will respond to *generic move set* messages by continuously transitioning at a speed which is calculated from the delta level field and another message field called Transition Time, optionally with a specified delay before starting. The transition will cease when a stop message is received containing a delta level of zero.

The Generic Move Set message types

The code required to send a *generic move set unacknowledged* message only differs from the code used when sending an acknowledged *generic move set* message in that the message opcode is different. Consequently we'll use a common function for both message types, with the required message opcode as a parameter. Add the following function in the section headed *Generic Level Client - TX message producer functions*:

```
void generic_move_set(s16_t delta_level, u8_t transition_time, u8_t delay, u32_t opcode)
{
    // 2 bytes for the opcode
    // 5 bytes of parameters / payload : 16 bits delta level,
    // 8 bits TID, 8 bits transition time, 8 bits delay.
    // 4 additional bytes for the TransMIC
    NET_BUF_SIMPLE_DEFINE(msg, 2 + 5 + 4);
    struct bt_mesh_msg_ctx ctx = {
        .net_idx = net_idx,
        .app_idx = app_idx,
        .addr = target,
        .send_ttl = BT_MESH_TTL_DEFAULT,
    };
    bt_mesh_model_msg_init(&msg, opcode);
    net_buf_simple_add_le16(&msg, delta_level);
}
```

```

net_buf_simple_add_u8(&msg, tid);
net_buf_simple_add_u8(&msg, transition_time);
net_buf_simple_add_u8(&msg, delay);
tid++;
if (bt_mesh_model_send(&sig_models[3], &ctx, &msg, NULL, NULL))
{
    printk("Unable to send generic move set (ack/unack) message\n");
}
printk("move set (ack/unack) message %d sent\n", delta_level);
}

```

Explanation

The function takes four parameters, the first of which is the delta level state value to send. The fourth is an opcode which will indicate which of *generic move set* or *generic move set unacknowledged* we wish to send. The other two parameters, *transition_time* and *delay* are available in other message types, such as *generic level set* but we haven't used them so far. In implementing the *generic move set* messages, we're taking the opportunity to explore the way these parameters can be used.

Field	Size (octets)	Notes
Delta Level	2	The Delta Level step to calculate Move speed for the Generic Level state.
TID	1	Transaction Identifier
Transition Time	1	Format as defined in Section 3.1.3 (optional).
Delay	1	Message execution delay in 5 milliseconds steps (C.1).

C.1: If the Transition Time field is present, the Delay field shall also be present; otherwise these fields shall not be present.

Table 3.42: Generic Move Set message parameters

Once again we use *NET_BUF_SIMPLE_DEFINE* to create a network buffer. We then create a message context and complete the initialisation of the required PDU using Zephyr API functions. We add the level state and TID (Transaction Identifier) values followed by the *transition_time* and *delay* values. We then increment the TID in readiness for sending our next message and send the message using the Zephyr *bt_mesh_model_send* function.

The Delay field is simple to understand. It specifies a period of time for which the receiving model will wait before executing the operation represented by the message. Its value represents a multiple of 5 milliseconds though, not the absolute time value itself.

Transition Time is a bit more involved. Here's the format from the specification:

Field	Size (bits)	Definition
Default Transition Number of Steps	6	The number of Steps
Default Transition Step Resolution	2	The resolution of the Default Transition Number of Steps field

Table 3.3: Generic Default Transition Time state format

The 8 bits of the Transition Time field are divided into 2 sub-fields, "number of steps" and "step resolution".

Step Resolution tells us the time value to attribute to each *step* (we'll come back to that word shortly). Its four possible values mean the value of one step is 100 ms (0b00), 1 second (0b01), 10 seconds (0b10) or 10 minutes (0b11). So if the Number of Steps field contains 10 and Step Resolution is 0b01 then this means the entire state transition should take 10 seconds to complete. If our message contains these values but also has a Delay field containing 200 then the transition will start executing one second after the message was received and complete 10 seconds later, making it a total of 11 seconds.

The word *step* suggests the size and number of the increments we might take as we transition from the current level by the specified delta level, but that's not quite what is intended. Using the *Number of Steps* and *Step Resolution* values, we calculate the speed at which the transition is executed but it can be performed in a completely continuous manner if required or in fact, via a series of discreet steps. This is an implementation detail. Basically, we use this data to set up a timer which will execute the required level transition in a given time period. See Figure 4 for an illustration of these concepts.

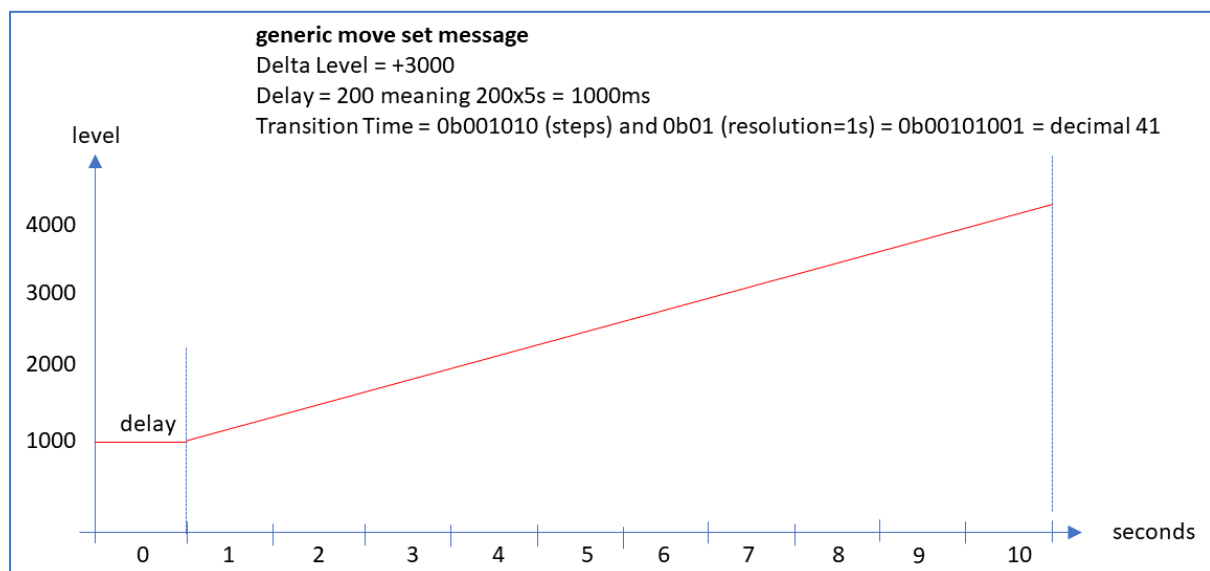


Figure 1 - delay and transition time fields

Checkpoint

We'll perform a couple of quick tests now by implementing some calls to our *generic_move_set* function with various parameters.

Generic Move Set

Update `buttonA_work_handler` as shown:

```
void buttonA_work_handler(struct k_work *work)
{
    printk("buttonA_work_handler\n");
    // set unack random
    // s16_t level = get_random_level();
    // generic_level_set(level,BT_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK);

    // set random
    // s16_t level = get_random_level();
    // generic_level_set(level,BT_MESH_MODEL_OP_GEN_LEVEL_SET);

    // delta set positive
```

```

// generic_delta_set(mapDimmerSettingToDelta(1),BT_MESH_MODEL_OP_GEN_DELTA_SET);

// delta set unack positive
// generic_delta_set(mapDimmerSettingToDelta(1),BT_MESH_MODEL_OP_GEN_DELTA_SET_UNACK);

// move set positive
// increase by 32767 with duration 2.5s and
// with a 1 second delay (200x5ms)
// transition time is 0b011001 (steps) then
// 0b00 (resolution=100ms) = 0b00011001 = decimal 25
generic_move_set(MAX_DELTA_LEVEL_16, 25, 200, BT_MESH_MODEL_OP_GEN_MOVE_SET);
}

```

When button A is pressed, it will send a *generic move set* message with a delta level value of +32767 which will start to execute after a 1000ms delay and will take 25 (steps) * 100ms = 2.5 seconds to complete a single pass through the specified delta.

Modify buttonB_handler so that it sends a *generic move set* message with a delta level value of +1.

```

void buttonB_work_handler(struct k_work *work)
{
    printk("buttonB work_handler\n");
    // generic_level_get();

    // delta set negative
    // generic_delta_set(mapDimmerSettingToDelta(-1),BT_MESH_MODEL_OP_GEN_DELTA_SET);

    // delta set unack negative
    // generic_delta_set(mapDimmerSettingToDelta(-1),BT_MESH_MODEL_OP_GEN_DELTA_SET_UNACK);

    // move set negative
    // decrease by 32767 with duration 2.5s and
    // with a 1 second delay (200x5ms)
    // transition time is 0b011001 (steps) then
    // 0b00 (resolution=100ms) = 0b00011001 = decimal 25
    generic_move_set(-1 * MAX_DELTA_LEVEL_16, 25, 200, BT_MESH_MODEL_OP_GEN_MOVE_SET);
}

```

When button B is pressed, it will send a *generic move set* message with a delta level value of -32767 which will start to execute after a 1000ms delay and will take 25 (steps) * 100ms = 2.5 seconds to complete a single pass through the specified delta.

Build and install the binary on your dimmer node. Reset both the dimmer and light nodes by pressing the reset button. Test by pressing button A and watch as the light node increases its level in a series of dynamic transitions from completely off (no LEDs lit) to fully on (all LEDs lit). Now press button B to reverse the process. Monitor the process using serial consoles attached to both the dimmer and light nodes over USB, remembering that we're sending acknowledged messages and therefore the dimmer node should receive status messages in response to each *generic move set* message it sends.

Modidy your code so that button B will stop a running move transition by sending a message with delta_level = 0 like this and re-test.

```

// move set unack with zero delta level to stop a continuously running move on the light
generic_move_set(0, 25, 200, BT_MESH_MODEL_OP_GEN_MOVE_SET_UNACK);

```

Generic Move Set Unacknowledged

Update buttonA_work_handler as shown:

```

void buttonA_work_handler(struct k_work *work)
{

```

```

printf("buttonA_work_handler\n");
// set unack random
// s16_t level = get_random_level();
// generic_level_set(level,BT_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK);

// set random
// s16_t level = get_random_level();
// generic_level_set(level,BT_MESH_MODEL_OP_GEN_LEVEL_SET);

// delta set positive
// generic_delta_set(mapDimmerSettingToDelta(1),BT_MESH_MODEL_OP_GEN_DELTA_SET);

// delta set unack positive
// generic_delta_set(mapDimmerSettingToDelta(1),BT_MESH_MODEL_OP_GEN_DELTA_SET_UNACK);

// move set positive
// increase by 32767 in 25 steps, each with duration 100ms and
// with a 1 second delay (200x5ms)
// transition time is 0b011001 (steps) then
// 0b00 (resolution=100ms) = 0b00011001 = decimal 25
// generic_move_set(MAX_DELTA_LEVEL_16, 25, 200, BT_MESH_MODEL_OP_GEN_MOVE_SET);

// move set unack positive
generic_move_set(MAX_DELTA_LEVEL_16, 25, 200, BT_MESH_MODEL_OP_GEN_MOVE_SET_UNACK);
}

```

And update buttonB_work_handler to send positive, unacknowledged generic delta set messages like this:

```

void buttonB_work_handler(struct k_work *work)
{
    printf("buttonB_work_handler\n");
    // generic_level_get();

    // delta set negative
    // generic_delta_set(mapDimmerSettingToDelta(-1),BT_MESH_MODEL_OP_GEN_DELTA_SET);

    // delta set unack negative
    // generic_delta_set(mapDimmerSettingToDelta(-1),BT_MESH_MODEL_OP_GEN_DELTA_SET_UNACK);

    // move set negative
    // decrease by 32767 in 25 steps, each with duration 100ms and
    // with a 1 second delay (200x5ms)
    // transition time is 0b011001 (steps) then
    // 0b00 (resolution=100ms) = 0b00011001 = decimal 25
    // generic_move_set(-1 * MAX_DELTA_LEVEL_16, 25, 200, BT_MESH_MODEL_OP_GEN_MOVE_SET);

    // move set unack negative
    generic_move_set(-1 * MAX_DELTA_LEVEL_16, 25, 200, BT_MESH_MODEL_OP_GEN_MOVE_SET_UNACK);
}

```

Checkpoint

Build, install and test again, remembering to reset both the dimmer and light micro:bits to ensure SEQ numbers do not get repeated (see Troubleshooting section). The test results should be as for the acknowledged *generic move set* messages except that status messages will not be returned.

Next

You've implemented a node to act as a dimmer control. Move on to the next exercise and implement the light node.