



Bluetooth Mesh Developer Study Guide

Bluetooth Mesh - Hands-on Coding Lab - On Off Switch Node

Version: 1.0.0

Last updated: 15th June 2018

Contents

REVISION HISTORY	4
EXERCISE 5 – IMPLEMENTING THE LIGHT	5
Introduction	5
Project Set Up	5
Start Point	5
The Generic OnOff Server Model	6
Messages.....	6
Generic OnOff Get and Generic OnOff Status	6
Explanation	6
Remaining Time Calculation.....	7
Explanation	7
Generic OnOff Status	7
Explanation	8
Testing.....	8
Generic OnOff Set and Generic OnOff Set Unacknowledged	9
Explanation	9
OnOff Work Handler	10
Testing.....	10
The Generic Level Model - Absolute Level Changes	11
Generic Level Get and Generic Level Status	11
Explanation	11
Generic Level Status.....	11
Explanation	12
Testing.....	12
Generic Level Set and Generic Level Set Unacknowledged	13
Explanation	14
Level Work Handler.....	14
Testing.....	14
The Generic Level Model - Relative Level Changes	15
Generic Delta Set and Generic Delta Set Unacknowledged	15
Explanation	16
Testing.....	16

The Generic Level Model - Move Level Changes	16
Understanding Transition Time and Delay	17
Generic Move Set and Generic Move Set Unacknowledged	17
Calculating Transition Times	18
Explanation	19
The generic move set functions	19
Explanation	21
Move Transition Timer	21
Explanation	21
Explanation	22
Testing.....	22
Conclusion	22

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	<p>generic move set was incorrectly described and implemented. Specifically, generic move set transitioned generic level through a fixed delta. The Delta Level field in move set messages must only be used in calculating the transition speed. There is no concept of a target level in move operations.</p> <p>This has been rectified.</p>

Exercise 5 – Implementing the Light

Introduction

Our light node will consist of the *generic onoff server model* and the *generic level server model*. A full implementation of these two models, compliant with the specification is not a small piece of work and is more than we need to concern ourselves with to meet the educational goals of this self-study resource. Consequently we'll be implementing only those aspects of the two server models that are required to support the switch and dimmer nodes we implemented in the previous two exercises.






Project Set Up

Create the following directories for your project:

```
light/  
  build/  
  src/
```

Copy all of the files and the src directory from \$MDG\code\start_state\Light\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 light 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.

Start Point

The starter code for the light node contains more code than was the case for the switch and dimmer. Code which we've now dealt with several times, like declaring security keys to support self-provisioning, address values, configuration client and server model declarations, health server model definitions and the steps involved in defining node composition are already in place. There's nothing to be learned from copying and pasting that code again and you have plenty of other, more interesting work to do in this exercise.

The Generic OnOff Server Model

The generic onoff server model supports the messages which our switch node will be sending.

The following figure, taken from the mesh model specification shows the messages that the generic onoff server model must support. We need each of these message types to support the functionality of our switch node and so shall be implementing all of them.

Element	SIG Model ID	States	Messages	Rx	Tx
Main	0x1000	Generic OnOff (see Section 3.1.1)	Generic OnOff Get	M	
			Generic OnOff Set	M	
			Generic OnOff Set Unacknowledged	M	
			Generic OnOff Status		M

Table 3.86: Generic OnOff Server elements, states, and messages

Messages

Generic OnOff Get and Generic OnOff Status

Our first task is to implement the processing of *generic onoff get* messages, which will necessitate being able to send *generic onoff status* messages.

Find the function *generic_onoff_get* in your code and update it so it looks like this:

```
static void generic_onoff_get(struct bt_mesh_model *model, struct bt_mesh_msg_ctx *ctx,
struct net_buf_simple *buf)
{
    printk("gen_onoff_get\n");
    generic_onoff_status(onoff_state, ctx->addr, transition_in_progress, target_onoff_state,
        calculate_remaining_time());
}
```

Explanation

This code is quite simple. We simply call another function, which will formulate and send a *generic onoff status* message. The parameters passed to this function are:

onoff_state	The current onoff state value
ctx->addr	The address of the remote device from which the <i>generic onoff get</i> message was received and that we are replying to.
transition_in_progress	A flag which indicates whether or not there's currently a state transition in progress.
target_onoff_state	The onoff state which the node will be in after the transition (if one is in progress) completes.
calculate_remaining_time()	The time remaining to complete the active transition, if one is in progress.

Remaining Time Calculation

In the previous exercise in the section on the *generic move* set message, we learned that the *remaining* time field consists of two sub-fields, Number of Steps and Step Resolution. With that in mind, let's now implement the *calculate_remaining_time()* function:

```
static u8_t calculate_remaining_time() {
    if (transition_in_progress == 0) {
        return 0;
    }
    s64_t now = k_uptime_get();
    u32_t duration_remainder = total_transition_duration - (now -
transition_start_timestamp);
    u8_t steps = 0;
    u8_t resolution = 0;
    if (duration_remainder > 620000) {
        // > 620 seconds -> resolution=0b11 [10 minutes]
        resolution = 0x03;
        steps = duration_remainder / 600000;
    } else if (duration_remainder > 62000) {
        // > 62 seconds -> resolution=0b10 [10 seconds]
        resolution = 0x02;
        steps = duration_remainder / 10000;
    } else if (duration_remainder > 6200) {
        // > 6.2 seconds -> resolution=0b01 [1 seconds]
        resolution = 0x01;
        steps = duration_remainder / 1000;
    } else {
        // <= 6.2 seconds -> resolution=0b00 [100 ms]
        resolution = 0x00;
        steps = duration_remainder / 100;
    }
    printk("calculated steps=%d,resolution=%d\n",steps,resolution);
    return ((resolution << 6) | steps);
}
```

This code needs to go somewhere above the *generic_onoff_get* function.

Explanation

If there's no timed state transition in progress, this code just returns zero and we won't start to deal with timed state transitions until we implement the generic move set message types. Other RX messages processed by this node could contain transition time and delay fields, defining timed state transitions and a specification compliant implementation would therefore, have to handle this eventuality. Our goal is to learn, not implement a product here and so we won't be handling timed state transitions other than in the generic move set case and so even though we're taking the opportunity to set this code up now, it will not yet be used because the *transition_in_progress* flag will not be set.

In those cases where the *transition_in_progress* flag is set however, you can see that we simply choose a resolution according to the magnitude of the *duration_remainder* variable, which will contain the amount of time left to complete a timed state transition and calculate the *number of steps* accordingly.

Generic OnOff Status

Add the following function under the *generic onoff status TX message producer* comment:

```
void generic_onoff_status(u8_t present_on_or_off, u16_t dest_addr, u8_t transitioning, u8_t
target_on_or_off, u8_t remaining_time)
{
    // 2 bytes for the opcode
    // 1 bytes parameters: present onoff value
    // 2 optional bytes for target onoff and remaining time
```

```

// 4 additional bytes for the TransMIC

struct bt_mesh_msg_ctx ctx = {
    .net_idx = net_idx,
    .app_idx = app_idx,
    .addr = dest_addr,
    .send_ttl = BT_MESH_TTL_DEFAULT,
};

u8_t buflen = 7;
if (transitioning == 1) {
    buflen = 9;
}

NET_BUF_SIMPLE_DEFINE(msg, buflen);

bt_mesh_model_msg_init(&msg, BT_MESH_MODEL_OP_GENERIC_ONOFF_STATUS);
net_buf_simple_add_u8(&msg, present_on_or_off);

if (transitioning == 1) {
    net_buf_simple_add_u8(&msg, target_on_or_off);
    net_buf_simple_add_u8(&msg, remaining_time);
}

if (bt_mesh_model_send(&sig_models[3], &ctx, &msg, NULL, NULL))
{
    printk("Unable to send generic onoff status message\n");
}

printk("onoff status message %d sent\n", present_on_or_off);
}

```

Explanation

This code formulates a *generic onoff status* message and sends it, using Zephyr APIs. All of these messages must contain the field *Present OnOff*. If a timed state transition is in progress, the function also includes the optional fields *Target OnOff* and *Remaining Time*.

Testing

Build and install the code for your switch node so that button B sends a *generic onoff get* or alternatively, install the ready made binary *onoff_client_set_unack_on_get.hex* from the `binaries/switch` folder.

Build and install your work-in-progress light node code from this exercise on another of your devices.

Press button B on your switch. Your switch node should briefly display a “0”, which is the onoff state value of your light node being returned as a *generic onoff status* message in response to the *generic onoff get* message sent by button B. Monitor the serial console of each device and if you’re not getting the result you expect, ensure you have enabled debugging for the light node in the project’s `prj_bbc_microbit.conf` file:

```

CONFIG_BT_DEBUG_LOG=y
CONFIG_BT_MESH_DEBUG_ACCESS=y
CONFIG_BT_MESH_DEBUG=y

```

Your light node should be logging messages like these:

```

[bt] [DBG] bt_mesh_model_rcv: (0x20001374) app_idx 0x0000 src 0x0001 dst 0xc000
[bt] [DBG] bt_mesh_model_rcv: (0x20001374) len 2: 8201
[bt] [DBG] bt_mesh_model_rcv: (0x20001374) OpCode 0x00008201
gen_onoff_get
[bt] [DBG] model_send: (0x20001374) net_idx 0x0000 app_idx 0x0000 dst 0x0001
[bt] [DBG] model_send: (0x20001374) len 3: 820400

```



```
onoff status message 0 sent
```

If so, then both *generic onoff get* and *generic onoff status* are working.

Generic OnOff Set and Generic OnOff Set Unacknowledged

Handling these two message types varies only in that the first type requires us to send a *generic onoff status* message as a response whereas the second does not.

Update the following two functions:

```
static void generic_onoff_set(struct bt_mesh_model *model, struct bt_mesh_msg_ctx *ctx,
struct net_buf_simple *buf)
{
    remote_addr = ctx->addr;
    set_onoff_state(model, ctx, buf);
    //respond with STATUS message
    k_work_submit(&onoff_status_work);
}

static void generic_onoff_set_unack(struct bt_mesh_model *model, struct bt_mesh_msg_ctx
*ctx, struct net_buf_simple *buf)
{
    remote_addr = 0;
    set_onoff_state(model, ctx, buf);
}
```

And add the *set_onoff_state* function:

```
static void set_onoff_state(struct bt_mesh_model *model, struct bt_mesh_msg_ctx *ctx,
struct net_buf_simple *buf)
{
    u8_t buflen = buf->len;
    target_onoff_state = net_buf_simple_pull_u8(buf);
    u8_t tid = net_buf_simple_pull_u8(buf);
    printk("set onoff state: onoff=%u TID=%u\n", target_onoff_state, tid);

    transition_time = 0;
    if (buflen > 2) {
        printk("message contains transition_time field - processing not implemented");
    }

    delay = 0;
    if (buflen > 3) {
        printk("message contains delay field - processing not implemented");
    }

    s64_t now = k_uptime_get();
    printk("%lld> starting worker\n", now);
    k_work_submit(&onoff_set_work);
}
```

Explanation

generic_onoff_set_unack simply calls a function *set_onoff_state* to set the onoff state to the value in the received message. For full compliance, we'd handle the message via an immediate state transition or a timed transition depending on the presence or absence of the *delay* and *transition time* fields. For the purposes of these exercises, we'll only concern ourselves with immediate on/off state transitions.

generic_onoff_set does all of those things too, but finishes by queuing a task (*k_work_submit*) which will send a *generic onoff status* message as soon as the processing of the *set* operation has completed.

`set_onoff_state` extracts the fields in the received message and then submits a task to execute the state transition into a Zephyr work queue, also using `k_work_submit`. We're using a work queue so that we can serialise related actions like execution of the state change itself and the possible sending of a status message afterwards. When we're dealing with immediate state transitions, this is not strictly necessary but for timed transitions, where we want to send the status message after the transition completes, it's a useful pattern. We'll do some work with timed transitions later on so we'll use this pattern throughout, whether or not we're coding for the timed transition case for a given message type or not.

OnOff Work Handler

The state transition task and status message response task we posted into a Zephyr work queue each have an associated function which processes them as they're taken off the work queue by the Zephyr framework, one at a time. They were set up in the main function with the following Zephyr API call:

```
k_work_init(&onoff_set_work, onoff_work_handler);
k_work_init(&onoff_status_work, onoff_work_handler);
```

As you can see, both types of task are handled by the same function. Currently, that function is empty so let's rectify that now:

```
void onoff_work_handler(struct k_work *work)
{
    s64_t now = k_uptime_get();

    if (work == &onoff_set_work) {
        printk("%lld> handling an onoff set work item\n", now);
        // we're not implemented timed transitions otherwise we might pause here to
        // handle the delay field
        now = k_uptime_get();
        printk("%lld> transitioning onoff state to %d\n", now, target_onoff_state);
        onoff_state = target_onoff_state;
        transition_in_progress = 0;
        struct mb_display *disp = mb_display_get();
        mb_display_image(disp, MB_DISPLAY_MODE_DEFAULT, K_FOREVER, &led_patterns[(onoff_state *
25)], 1);
        return;
    }

    if (work == &onoff_status_work) {
        printk("%lld> sending an onoff status work item to %d\n", now, remote_addr);
        generic_onoff_status(onoff_state, remote_addr, 0, 0, 0);
        return;
    }
}
```

Testing

Build and install your code on the light node. Install the `onoff_client_set_on_set_off.hex` binary on your switch node or use your own code with button A set to send a *generic onoff set (0)* message and button B set to send a *generic onoff set(1)* message.

Pressing button B on your switch should result in all LEDs lit on the light and a "1" displayed briefly on the switch. Pressing A on the switch should switch all LEDs off on the light and cause a "0" to be displayed briefly on the switch.

The Generic Level Model - Absolute Level Changes

Once again, to simplify our work, we will only concern ourselves with those aspects of these messages that involve immediate rather than timed level state transitions since this is what we did in exercise 4. Our concern is to handle messages which get or set the *generic level* state. To reflect the generic level state value, we'll light some number of the 25 LEDs in proportion.

The messages we will implement support for are: *generic level get*, *generic level set*, *generic level set unack* and *generic level status*.

Generic Level Get and Generic Level Status

Our first task is to implement the processing of *generic level get* messages, which will necessitate being able to send *generic level status* messages.

Find the function *generic_level_get* in your code and update it so it looks like this:

```
static void generic_level_get(struct bt_mesh_model *model, struct bt_mesh_msg_ctx *ctx,
struct net_buf_simple *buf)
{
    remote_addr = ctx->addr;
    generic_level_status(remote_addr);
}
```

Explanation

This code is quite simple. We simply call another function, *generic_level_status* which will formulate and send a *generic level status* message. The parameters passed to this function are:

remote_addr	The address of the remote device from which the <i>generic onoff get</i> message was received and that we are replying to.
-------------	--

Generic Level Status

Update the following function so that it looks like this:

```
void generic_level_status(u16_t dest_addr)
{
    struct bt_mesh_msg_ctx ctx = {
        .net_idx = net_idx,
        .app_idx = app_idx,
        .addr = dest_addr,
        .send_ttl = BT_MESH_TTL_DEFAULT,
    };
    u8_t params_len = 2;
    printk("generic_level_status transition_in_progress=%d\n", transition_in_progress);
    if (transition_in_progress == 1) {
        params_len = 5;
    }

    NET_BUF_SIMPLE_DEFINE(msg, 2 + params_len + 4);
    bt_mesh_model_msg_init(&msg, BT_MESH_MODEL_OP_GENERIC_LEVEL_STATUS);

    printk("generic_level_status transition_in_progress=%d\n", transition_in_progress);
    if (transition_in_progress == 0) {
        // 2 bytes for the opcode
        // 2 bytes parameters: present level value
        // 4 additional bytes for the TransMIC
        net_buf_simple_add_le16(&msg, level_state);
    }
}
```

```

} else {
    // 2 bytes for the opcode
    // 5 bytes parameters: present level value, target level value, remaining time
    // 4 additional bytes for the TransMIC
    net_buf_simple_add_le16(&msg, level_state);
    net_buf_simple_add_le16(&msg, target_level_state);
    net_buf_simple_add_u8(&msg, calculate_remaining_time());
}

if (bt_mesh_model_send(&sig_models[4], &ctx, &msg, NULL, NULL))
{
    printk("Unable to send generic level status message\n");
}

s64_t now = k_uptime_get();
printk("%lld> level status message %d sent\n",now,level_state);
}

```

Explanation

This code formulates a *generic level status* message and sends it, using Zephyr APIs. All of these messages must contain the field *Present Level*. If a timed state transition is in progress, the function also includes the optional fields *Target Level* and *Remaining Time*. We already have the function required to derive the *remaining time* field.

Testing

Build and install the code for your dimmer node so that button B sends a *generic level get* or alternatively, install the ready made binary *level_client_A_set_unack_random_B_get.hex* from the binaries/dimmer folder.

Build and install your work-in-progress light node code from this exercise on another of your devices.

Monitor the serial console of each device.

Press button B on the dimmer node. The console for the dimmer node should indicate that a *generic level get* message has been sent and soon after, that a *generic level status* message was received. When the status message is received, the level value it contains should be displayed briefly on the micro:bit. In this case it should be 0.

If you're not getting the result you expect, ensure you have enabled debugging for the light node in the project's `prj_bbc_microbit.conf` file:

```

CONFIG_BT_DEBUG_LOG=y
CONFIG_BT_MESH_DEBUG_ACCESS=y
CONFIG_BT_MESH_DEBUG=y

```

Your light node should be logging messages like these:

```

generic_level_get
generic_level_status transition_in_progress=0
generic_level_status transition_in_progress=0
12312> level status message 0 sent

```

If so, then both *generic level get* and *generic level status* are working.

Generic Level Set and Generic Level Set Unacknowledged

Handling these two message types varies only in that the first type requires us to send a *generic level status* message as a response whereas the second does not.

Update the following two functions:

```
static void generic_level_set(struct bt_mesh_model *model, struct bt_mesh_msg_ctx *ctx,
struct net_buf_simple *buf)
{
    remote_addr = ctx->addr;
    generic_level_set_common(model, ctx, buf);
}

static void generic_level_set_unack(struct bt_mesh_model *model, struct bt_mesh_msg_ctx
*ctx, struct net_buf_simple *buf)
{
    printk("generic_level_set_unack\n");
    remote_addr = 0;
    generic_level_set_common(model, ctx, buf);
}
```

And add the *generic_level_set_common* function:

```
static void generic_level_set_common(struct bt_mesh_model *model, struct bt_mesh_msg_ctx
*ctx, struct net_buf_simple *buf)
{
    printk("generic_level_set_common\n");
    u8_t buflen = buf->len;
    // Level(2), TID(1), Transition Time(optional, 1), Delay (conditional, 1)
    target_level_state = (s16_t) net_buf_simple_pull_le16(buf);

    // The TID field is a transaction identifier indicating whether the message is a new
    message or a retransmission of a previously sent message
    u8_t tid = net_buf_simple_pull_u8(buf);

    // set the Generic Level state to the Level field of the message, unless the message has
    the same values for the SRC, DST, and TID fields as the
    // previous message received within the last 6 seconds.

    s64_t now = k_uptime_get(); // elapsed time since the system booted, in milliseconds.
    if (ctx->addr == last_message_src && ctx->recv_dst == last_message_dst && tid ==
last_message_tid && (now - last_message_timestamp <= 6000)) {
        printk("Ignoring message - same transaction during 6 second window\n");
        return;
    }
    last_message_timestamp = now;
    last_message_src = ctx->addr;
    last_message_dst = ctx->recv_dst;
    last_message_tid = tid;
    transition_time = 0;
    if (buflen > 3) {
        printk("message contains transition_time field - processing not implemented");
    }
    delay = 0;
    if (buflen > 4) {
        printk("message contains delay field - processing not implemented");
    }
    set_level_state(target_level_state);
    if (remote_addr != 0) {
        k_work_submit(&level_status_work);
    }
}
```

Explanation

generic_level_set_unack simply calls a function *generic_level_set_common* to set the level state to the value in the received message.

generic_level_set does the same but finishes by queuing a task (*k_work_submit*) which will send a *generic level status* message as soon as the processing of the *set* operation has completed if the *remote_addr* variable is non-zero, as set by the *generic_level_set* function. We haven't implemented the work handler which will send the status message yet. We'll do that shortly.

generic_level_set_common extracts the fields in the received message and sets the state variable accordingly.

Level Work Handler

We're using a Zephyr work queue to handle status message responses and we'll also use it for timed transition processing later in these exercises. They were set up in the main function with the following Zephyr API calls:

```
k_work_init(&level_set_work, level_work_handler);
k_work_init(&level_status_work, level_work_handler);
```

As you can see, both types of task are handled by the same function. Currently, that function is empty so let's rectify that now:

```
void level_work_handler(struct k_work *work)
{
    s64_t now = k_uptime_get();
    if (work == &level_set_work) {
        printk("%lld> handling an level set work item\n", now);
        printk("sleeping: delay %d ms\n", (delay*5));
        k_sleep(delay*5);
        now = k_uptime_get();
        printk("%lld> transitioning level state to %d\n", now, target_level_state);
        k_work_submit(&level_transition_work);
        return;
    }

    if (work == &level_status_work) {
        printk("%lld> sending a level status work item to %d\n", now, remote_addr);
        generic_level_status(remote_addr);
        return;
    }
}
```

We're only concerned with the *level_status_work* case at this stage.

Testing

Build and install your code on the light node. Install the *level_client_A_set_random_B_get.hex* binary on your dimmer node or use your own code with button A set to send a *generic level set (random value)* message and button B set to send a *generic level get()* message.

Pressing button A on your switch should result in a random number of LEDs being lit on the light. You should also receive a status message at the dimmer node.

The Generic Level Model - Relative Level Changes

Having implemented support for absolute level changes, we'll now focus on delta messages which allow us to set the level of remote nodes to a relative value. As you'd expect, there's a lot involved in implementing this set of capabilities that is very similar or identical to work you've previously done. And once again, we will only concern ourselves with those aspects of these messages that involve immediate rather than timed level state transitions. We'll progress a bit more quickly rather than review each step in detail, therefore.

The messages we will implement support for are: *generic delta set* and *generic delta set unack*.

Generic Delta Set and Generic Delta Set Unacknowledged

The pattern here is identical to that which we've followed in earlier exercises (e.g. for *generic level set* and *generic level set unacknowledged*). Add the following functions, overwriting the skeleton definitions of *generic_delta_set* and *generic_delta_set_unack* which are already there.

```
static void generic_delta_set_common(struct bt_mesh_model *model, struct bt_mesh_msg_ctx
*ctx, struct net_buf_simple *buf)
{
    // Delta(4), TID(1), Transition Time(optional, 1), Delay (conditional, 1)
    u8_t buflen = buf->len;
    delta_level = (s32_t) net_buf_simple_pull_le32(buf);
    u8_t tid_field = net_buf_simple_pull_u8(buf);
    if (tid_field != last_delta_tid) {
        // new transaction
        initial_trans_level = level_state;
    }

    s64_t now = k_uptime_get(); // elapsed time since the system booted, in milliseconds.

    transition_time = 0;

    if (buflen > 5) {
        printk("message contains transition_time field - processing not implemented");
    }
    delay = 0;
    if (buflen > 6) {
        printk("message contains delay field - processing not implemented");
    }

    if (ctx->addr == last_message_src && ctx->recv_dst == last_message_dst && tid_field ==
last_message_tid && (now - last_message_timestamp <= 6000)) {
        printk("Ignoring message - same transaction during 6 second window\n");
        return;
    }
    last_message_timestamp = now;
    last_message_src = ctx->addr;
    last_message_dst = ctx->recv_dst;
    last_message_tid = tid_field;

    s32_t new_level = level_state + delta_level;
    if (new_level < 0) {
        new_level = 0;
    } else if (new_level > MAX_LEVEL) {
        new_level = MAX_LEVEL;
    }
    target_level_state = new_level;

    set_level_state(target_level_state);
    if (remote_addr != 0) {
        k_work_submit(&level_status_work);
    }
}

static void generic_delta_set(struct bt_mesh_model *model, struct bt_mesh_msg_ctx *ctx,
struct net_buf_simple *buf)
{
    remote_addr = ctx->addr;
```

```

    generic_delta_set_common(model, ctx, buf);
}

static void generic_delta_set_unack(struct bt_mesh_model *model, struct bt_mesh_msg_ctx
*ctx, struct net_buf_simple *buf)
{
    remote_addr = 0;
    generic_delta_set_common(model, ctx, buf);
}

```

Explanation

The pattern here is identical to that which we have seen in previous cases. Differences lie in the details. In this case, rather than set the level state to the value provided in the set messages, we calculate a new level state as a delta to the current, capping at maximum and minimum values, an implementation detail which the specification allows.

Testing

Build and install your code on the light node. Install the `level_client_A_delta_set_pos_B_delta_set_neg.hex` binary on your dimmer node or use your own code with button A set to send a *generic delta set* message with a positive delta value and button B set to send a negative value.

Pressing button A repeatedly on your dimmer node should result in progressively more LEDs lighting on the light node. Pressing B should reduce the number of LEDs that are lit.

The Generic Level Model - Move Level Changes

Move operations are intended to allow dynamic transitions of state to take place at a rate determined by message parameters. They're similar to delta transitions except that a move operation, once initiated, can execute continuously until a special *stop message* is received. A *stop message* is a *generic move set* message which has the delta level field set to 0.

generic move set and *generic move set unacknowledged* messages have the following structure:

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

Transition Time is designated optional (and consequently, given its conditional relationship with Transition Time, so is the Delay field). This optionality is not what it seems at first glance, however. Move operations require a transition time but that transition time can be present in the set message itself or it can come from an optional configuration state called Generic Default Transition Time. If transition time is not available from either of these sources, the move operation may not be carried out.

We shall not implement support for the Generic Default Transition Time and therefore all *generic move set* and *generic move set unacknowledged* messages must contain Transition Time and Delay fields for our purposes.

The messages we will implement support for are: *generic move set* and *generic move set unack*.

Understanding Transition Time and Delay

You should consult the mesh models specification, section 3.1.3 for the precise definition of these fields. Delay is an 8 bit value which should be interpreted as a number of 5ms steps. Transition Time breaks down into 2 fields, with Step Resolution indicated by the 2 most significant bits and Num Steps the remaining 6 bits. These two fields allow us to calculate the total time the move state transition should take.

Examples can be useful, so here's one which should illustrate how these fields are interpreted.

Example move set message with transition time and delay fields present

field	octet values (hex)	meaning
opcode	820C	generic move set unacknowledged
delta level	FF7F	+32767
tid	00	transaction 0
transition time	19	0b00011001 where bits 6 and 7 are 0b00 = 0 Step Resolution = 0 meaning 100ms bits 0 to 6 are 0b011001 = 25 Num Steps = 25 transition time = 25 * 100ms = 2500ms, therefore
delay	C8	0xC8 = 200 decimal so this represents a delay of 200 x 5s = 1000ms

In plain English this means that the client is asking the server to change its level state by +32767. The transition time field is saying that the entire transition should be accomplished in 2500ms elapsed but the delay field says that the transition should not start until 1 second after the messages was received (200 * 5ms).

Generic Move Set and Generic Move Set Unacknowledged

Once again, the general pattern for receiving and processing messages is the one you should now be accustomed to. The major differences in how we handle the move messages is that we need to decode and then make use of the Transition Time and Delay fields. We'll use [Zephyr timer objects](#) to implement the dynamic nature of move operations.

Calculating Transition Times

Update the `derive_transition_time_values` function so that it reads as follows:

```
void derive_transition_time_values(u8_t transition_time)
{
    resolution = (transition_time >> 6);
    steps_multiplier = transition_time & 0x3F;
    printk("steps=%d,resolution=%d\n", steps_multiplier, resolution);
    // The number of Steps = 6 bits : 0x01-0x3E
    // Resolution = 2 bits = time taken for each transition step (e.g. 0b00 means 100ms,
    // 0b01 means 1 second)
    // so the elapsed transition time is Number of Steps * Resolution Meaning
    // Example
    // Delta Level = 1000
    // Steps = 100, Resolution = 100ms
    // After (delay * 5ms), every 100ms, level state is increased by (1000 / 100) = 10
    // until the complete level delta change has completed
    // calculate led_interval

    // how many LEDs will need to be switched on or off for this delta?
    delta_level_as_led_change_count = 0;
    if (delta_level != 0)
    {
        delta_level_as_led_change_count = (int)(abs(delta_level) /
level_one_step_value) + 1;
        if (delta_level_as_led_change_count > 25)
        {
            delta_level_as_led_change_count = 25;
        }
    }

    count_led_increments = 0;
    printk("delta_level_as_led_change_count=%d delta_level=%d
level_one_step_value=%d\n", delta_level_as_led_change_count, delta_level,
level_one_step_value);

    total_transition_duration = 0;

    switch (resolution)
    {
        // 100ms
        case 0:
            total_transition_duration = steps_multiplier * 100;
            break;
        // 1 second
        case 1:
            total_transition_duration = steps_multiplier * 1000;
            break;
        // 10 seconds
        case 2:
            total_transition_duration = steps_multiplier * 10000;
            break;
        // 10 minutes
        case 3:
            total_transition_duration = steps_multiplier * 600000;
            break;
    }

    led_interval = total_transition_duration / delta_level_as_led_change_count;

    printk("transition time=%d to switch %d LEDs each taking %d ms\n",
total_transition_duration, delta_level_as_led_change_count, led_interval);

    // only used by level operations. not relevant to onoff.
    one_led_level_increment = delta_level / delta_level_as_led_change_count;
    printk("delta_level=%d delta in terms of led increments=%d one led level increment
duration=%d\n", delta_level, delta_level_as_led_change_count, one_led_level_increment);
}
}
```

Explanation

The `transition_time` parameter contains the field of that name from a received mesh message and one of the things this function does is to split it into its two sub-fields of Step Resolution and Number of Steps. Given we're using the 25 LEDs of a micro:bit to represent 26 possible levels, from 0 (OFF) to 25 (FULLY ON), the function also calculates how many LEDs would change over the specified delta level and from this and the transition time, how quickly this should happen. The key line of code which calculates the transition speed is:

```
led_interval = total_transition_duration / delta_level_as_led_change_count;
```

The generic move set functions

Add the following function, somewhere suitable in your source.

```
static int generic_move_set_common(struct bt_mesh_model *model, struct bt_mesh_msg_ctx
*ctx, struct net_buf_simple *buf)
{
    printk("generic_move_set_common\n");
    u8_t buflen = buf->len;
    // Delta(2), TID(1), Transition Time(optional, 1), Delay (conditional, 1)
    delta_level = (s16_t)net_buf_simple_pull_le16(buf);
    u8_t tid_field = net_buf_simple_pull_u8(buf);
    if (tid_field != last_delta_tid)
    {
        // new transaction
        initial_trans_level = level_state;
        target_level_state = level_state + delta_level;
    }

    printk("buflen=%d delta_level=%d tid=%d\n", buflen, delta_level, tid_field);
    s64_t now = k_uptime_get(); // elapsed time since the system booted, in
milliseonds.
    if (ctx->addr == last_message_src && ctx->recv_dst == last_message_dst && tid_field
== last_message_tid && (now - last_message_timestamp <= 6000))
    {
        printk("Ignoring message - same transaction during 6 second window\n");
        return 1;
    }

    last_message_timestamp = now;
    last_message_src = ctx->addr;
    last_message_dst = ctx->recv_dst;
    last_message_tid = tid_field;
    transition_time = 0;
    steps_multiplier = 0;

    printk("delta_level=%d transition_in_progress=%d\n", delta_level,
transition_in_progress);
    // transition_in_progress will already have been set to 0 so we don't test it for
being equal to 1 here
    if (delta_level == 0)
    {
        printk("generic move set has requested running transition be stopped\n");
        if (k_timer_remaining_get(&level_transition_timer) == 0)
        {
            printk("stopping timer\n");
            k_timer_stop(&level_transition_timer);
        }
        // prevent the timer from being started for this case
        return 1;
    }

    if (buflen > 4)
    {
        transition_time = net_buf_simple_pull_u8(buf);
        delay = net_buf_simple_pull_u8(buf);
        printk("transition_time=%d\n", transition_time);
    }
}
```

```

        printk("delay=%d\n", delay);
        derive_transition_time_values(transition_time);
        return 0;
    }
    else
    {
        // we do not support the Generic Default Transition Time state and therefore
        require both the transition time field and the delay field
        // Consequently, per the spec, we shall not initiate any Generic Level state
        change
        printk("Ignoring message - transition time and delay are absent\n");
        return 1;
    }
}

```

Now, per our usual pattern, replace the skeleton implementation of the `generic_move_set` and `generic_move_set_unack` functions with the following:

```

static void generic_move_set(struct bt_mesh_model *model, struct bt_mesh_msg_ctx *ctx,
struct net_buf_simple *buf)
{
    if (transition_in_progress == 1)
    {
        k_timer_stop(&level_transition_timer);
        transition_in_progress = 0;
    }

    remote_addr = ctx->addr;
    // ref 3.3.2.2.4 Upon receiving a Generic Move Set message, the Generic Level Server
    shall respond
    // with a Generic Level Status message (see Section 3.3.2.2.5). The target Generic
    Level state is
    // the upper limit of the Generic Level state when the transition speed is positive,
    or the lower
    // limit of the Generic Level state when the transition speed is negative.

    // extract fields from message
    int err = generic_move_set_common(model, ctx, buf);
    if (err)
    {
        return;
    }
    // save the real target_level_state from the message
    s16_t msg_target_level_state = target_level_state;
    // set the target level state to the max/min value for the status message
    if (delta_level >= 0)
    {
        target_level_state = 32767;
    }
    else
    {
        target_level_state = -32768;
    }
    // send the status message
    generic_level_status(remote_addr);
    // restore the target level value
    target_level_state = msg_target_level_state;
    // kick off timer to control transition process
    transition_in_progress = 1;
    start_level_transition_timer();
}

static void generic_move_set_unack(struct bt_mesh_model *model, struct bt_mesh_msg_ctx
*ctx, struct net_buf_simple *buf)
{
    if (transition_in_progress == 1)
    {
        k_timer_stop(&level_transition_timer);
        transition_in_progress = 0;
    }
    remote_addr = 0;
    // extract fields from message
    int err = generic_move_set_common(model, ctx, buf);
    if (err)

```

```

    {
        return;
    }

    // kick off timer to control transition process
    transition_in_progress = 1;
    start_level_transition_timer();
}

```

Explanation

The *generic_move_set_common* function extracts fields from the received message and derives the time the level state transition must be executed over from the Transition Time field.

The rules for responding to *generic move set* messages are unusual in that the status message must be returned immediately rather than after the transition has completed. This is because a move might run indefinitely. Consequently, *generic_move_set* has a bit more work to do than usual, to extract fields from the received message, send back the *generic level status* message with the correct *target level* field value of either +32,767 or -32,768, depending on the direction of “travel” and then initiate the move transition itself. The unacknowledged message just extract fields from the received messages and then kicks off the transition.

If the *delta level* field contains zero then any transition which is already running is stopped.

Move Transition Timer

Add the *start_level_transition_timer* function:

```

void start_level_transition_timer() {
    transition_in_progress = 1;
    transition_start_timestamp = k_uptime_get();
    s64_t now = k_uptime_get();
    printk("%lld> k_timer_start called\n", now);
    k_timer_start(&level_transition_timer, K_MSEC(5 * delay), K_MSEC(led_interval));
}

```

Explanation

In *start_level_transition_timer* we set a flag to indicate that a transition is in progress. Take a look back at the code in the *generic_level_status* function. You’ll see that the format of the generic level status message is different, depending on whether or not there’s a transition already in progress. That’s where we use this flag.

We then kick off a Zephyr timer to control our progression through the state change using timing parameters from the received message. The *level_transition_timer* has an associated handler function which is very simple and will be called at intervals by the timer. All it does is to queue an item of work to be executed.

```

// this function is already defined in the starter code so no need to add it to your code

void level_transition_timer_handler(struct k_timer *dummy)
{
    k_work_submit(&level_transition_work);
}

```

The *level_transition_work object* is processed in a handler function called *level_transition-work_handler* and it was already written for you and in the starter code. Here it is:

```
// this function is already defined in the starter code so no need to add it to your code

void level_transition_work_handler(struct k_work *work)
{
    s64_t now = k_uptime_get();
    printk("%lld> target_level_state=%d level_state=%d\n", now, target_level_state,
level_state);
    // have we gone through all required transition steps or reached a limit?
    s32_t new_level = level_state + one_led_level_increment;
    if (new_level > MAX_LEVEL)
    {
        new_level = 0;
    }
    else if (new_level < 0)
    {
        new_level = MAX_LEVEL;
    }
    level_state = new_level;
    set_level_state(level_state);
    if ((transition_stop_requested == 1))
    {
        printk("Stopping transition timer\n");
        transition_in_progress = 0;
        k_timer_stop(&level_transition_timer);
    }
}
```

Explanation

The way we've implemented timed transitions is to divide a calculated transition time by 25, the number of LEDs which we have on the micro:bit display. The level transition code is driven by a timer and at each execution, increments the level state by the amount which corresponds to a single LED being switched on or off. When the maximum or minimum value for the *generic level* state is reached or exceeded, we wrap around and continue.

Testing

Build and install your code on the light node. Install the `level_client_A_move_set_unack_pos_B_move_set_unack_stop.hex` binary on your dimmer node or use your own code with button A set to send a *generic move set* message with a positive delta value of 32767 and button B set to send a delta value of 0 (which will act as a stop message).

Pressing button A **once** on your dimmer node should result in progressively more LEDs lighting on the light node being lit and the transition repeated, continuously. You should notice a delay before anything happens and then a smooth progression of more and more LEDs being lit until all 25 are illuminated, at which point all LEDs should be switched off and the transition repeated. Pressing B **once** should stop the transition.

Conclusion

This study guide has led you through the process of developing Bluetooth mesh code using the Zephyr RTOS as an example platform. If you've understood and completed the work in these exercises, then you're well on your way to being able to handle anything and everything you'll ever need to do to create Bluetooth mesh products.

For further details and information on other models you should consult the Bluetooth mesh profile, models and properties specifications which are available from <https://www.bluetooth.com/specifications/mesh-specifications>

Congratulations. Now, over to you.