Developer Study Guide: An introduction to Bluetooth Mesh Networking

Hands-on Coding Exercise - Generic On Off Server

Release	:	2.1.0
Document Version	:	2.1.0
Last updated	:	29th June 2021

Contents

Revision History

Version	Date	Author	Changes
1.0.0	15th June 2018	Martin Woolley Bluetooth SIG	Initial version.
1.0.2	16th August 2018	Martin Woolley Bluetooth SIG	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. This has been rectified.
1.0.4	14th December 2018	Martin Woolley Bluetooth SIG	Minor errata: #11 - Exercising the generic level state would work visibly even if the onoff state was OFF. This was not correct. Imagine a dimmer control which when pressed acts as an on/off switch. Rotating the knob will have no effect if the lights are switched off. That's how the generic onoff server and generic level server should work when incorporated together in a device. The two states are now handled completely independently. Code and documentation adjusted accordingly. #12 - Light node should have subscribed to the group address once for each model. A bug in Zephyr 1.12 allowed a subscription to only one model to be sufficient for all models to have messages published to that address routed to them so there was no user discernible impact of this issue at Zephyr 1.12. Code has been adjusted.
2.0.0	16th December 2019	Martin Woolley Bluetooth SIG	Exercises are now based on Zephyr 1.14 using the west multipurpose tool and a Nordic Thingy developer board.
2.0.1	21st December 2020	Martin Woolley Bluetooth SIG	Language changes
2.1.0	29th June 2021	Martin Woolley Bluetooth SIG	**Release:** Zephyr code now based on version 2.6.0 of the Zephyr SDK **Document**: Code fragments revised to be based on Zephyr 2.6.0

Exercise 4 – Implementing the Generic On Off Server

Introduction

Our light node includes an LED which we want to be able to switch on and off by sending messages from the developer board in which the *generic on off client model* was implemented in exercise 3. A full implementation of the server 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 generic onoff server model that are required to support the basic on off requirements of a simple switch. We shall not concern ourselves with behaviours such as timed state transitions, for example.

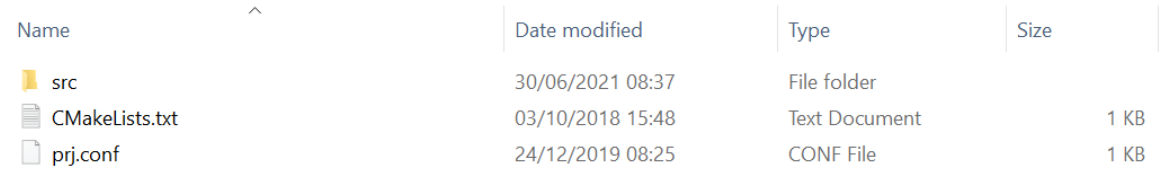
Project Set Up

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:



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

Prepare your project by executing the following commands. The argument thingy52_nrf52832 builds for a Nordic Thingy. If you are using a different developer board, adjust this command accordingly, with reference to the Zephyr SDK documentation.

```
west build -b thingy52_nrf52832
```

Your starter code should compile and link.

```
C:\mdk_work\Light>west build -b thingy52_nrf52832
-- west build: generating a build system
-- Application: C:/mdk_work/Light
-- Zephyr version: 2.6.0 (C:/workspaces/zephyr_source/zephyr), build: zephyr-v2.6.0
-- Found Python3: C:/python39/python.exe (found suitable exact version "3.9.2") found
components:  Interpreter
-- Found west (found suitable version "0.10.1", minimum required is "0.7.1")
-- Board: thingy52_nrf52832
-- Cache files will be written to: C:\Users\mwoolley\AppData\Local/.cache/zephyr
-- Found dtc: C:/ProgramData/chocolatey/bin/dtc.exe (found suitable version "1.4.7",
minimum required is "1.4.6")
-- Found toolchain: gnuarmemb (C:/gnu_arm_embedded)
```

```
-- Found BOARD.dts:
C:/workspaces/zephyr_source/zephyr/boards/arm/thingy52_nrf52832/thingy52_nrf52832.dts
-- Generated zephyr.dts: C:/mdk_work/Light/build/zephyr/zephyr.dts
-- Generated devicetree_unfixed.h:
C:/mdk_work/Light/build/zephyr/include/generated/devicetree_unfixed.h
-- Generated device_extern.h:
C:/mdk_work/Light/build/zephyr/include/generated/device_extern.h
Parsing C:/workspaces/zephyr_source/zephyr/Kconfig
Loaded configuration
'C:/workspaces/zephyr_source/zephyr/boards/arm/thingy52_nrf52832/thingy52_nrf52832_defconfi
g'
Merged configuration 'C:/mdk_work/Light/prj.conf'
Configuration saved to 'C:/mdk_work/Light/build/zephyr/.config'
Kconfig header saved to 'C:/mdk_work/Light/build/zephyr/include/generated/autoconf.h'
-- The C compiler identification is GNU 7.3.1
-- The CXX compiler identification is GNU 7.3.1
-- The ASM compiler identification is GNU
-- Found assembler: C:/gnu_arm_embedded/bin/arm-none-eabi-gcc.exe
-- Configuring done
-- Generating done
-- Build files have been written to: C:/mdk_work/Light/build
-- west build: building application
[260/267] Linking C executable zephyr\zephyr_prebuilt.elf

[267/267] Linking C executable zephyr\zephyr.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:      235696 B       512 KB     44.96%
            SRAM:       33293 B        64 KB     50.80%
        IDT_LIST:          0 GB         2 KB      0.00%
```

If you got errors then your Zephyr SDK is probably not installed and configured properly. Consult the Zephyr documentation or use the Zephyr mailing lists or Slack channel for help.

Start Point

The starter code for the light node contains more code than was the case for the switch node. Code for issues which we've already dealt with like health server model definitions and the main 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.

Node Composition

The basic node composition structure is already in place in the starter code but we still need to add the generic on off server model to our sole element.

The Generic OnOff Server Model

The generic onoff server model supports the generic on off messages which our switch node can send.

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

Generic OnOff Message Types

Under the comment *generic onoff server message opcodes*, add the following message opcode definitions:

```
#define BT_MESH_MODEL_OP_GENERIC_ONOFF_GET BT_MESH_MODEL_OP_2(0x82, 0x01)
#define BT_MESH_MODEL_OP_GENERIC_ONOFF_SET BT_MESH_MODEL_OP_2(0x82, 0x02)
#define BT_MESH_MODEL_OP_GENERIC_ONOFF_SET_UNACK BT_MESH_MODEL_OP_2(0x82, 0x03)
#define BT_MESH_MODEL_OP_GENERIC_ONOFF_STATUS BT_MESH_MODEL_OP_2(0x82, 0x04)
```

Explanation

We've defined constants for each of the message types that are part of the generic onoff server model and will be referencing them elsewhere in our code as we complete the node composition.

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 under the comment *generic onoff server functions*.

```
// need to forward declare as we have circular dependencies
void generic_onoff_status(bool publish, uint8_t on_or_off);

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");
}

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

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

static const struct bt_mesh_model_op generic_onoff_op[] = {
                {BT_MESH_MODEL_OP_GENERIC_ONOFF_GET, 0, generic_onoff_get},
                {BT_MESH_MODEL_OP_GENERIC_ONOFF_SET, 2, generic_onoff_set},
                {BT_MESH_MODEL_OP_GENERIC_ONOFF_SET_UNACK, 2, generic_onoff_set_unack},
                BT_MESH_MODEL_OP_END,
};
```

Explanation

The array of *bt_mesh_model_op* types maps the opcode of generic on off messages to functions which will handle messages of each type and indicates the minimum permitted access message payload length. BT_MESH_MODEL_END indicates the end of the definition.

A skeleton message handler function has been implemented in each case.

Completing Node Composition

Update your sig_models array so that it includes an entry for the *generic on off server model* and looks like this:

```
static struct bt_mesh_model sig_models[] = {
        BT_MESH_MODEL_CFG_SRV,
        BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
        BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_ONOFF_SRV, generic_onoff_op, &generic_onoff_pub,
NULL),
};
```

Add a message publication context for use by the *generic on off server model* under the comment *generic onoff server model publication context*.

```
// generic onoff server model publication context
BT_MESH_MODEL_PUB_DEFINE(generic_onoff_pub, NULL, 2 + 1);
```

Testing #1

Build and flash your generic on off server code to the Nordic Thingy by executing *west flash*.

Make sure you have switched the Thingy on with its on/off slider switch by the USB port

Plug in your switch node and ensure it is running the completed code from exercise 3 in the previous lab. Button 1 should send a *generic on off set unacknowledged (1),* button 2 should send a *generic on off set unacknowledged (0)* and button 3 should send a *generic on off get*.

Start the J-Link RTT Viewer so that you can see console messages generated by the Nordic Thingy.

Your light node should be logging messages like these:

```
00> *** Booting Zephyr OS build zephyr-v2.6.0  ***
00>
00> thingy light node v1.1.0
00>
00> GPIO_ACTIVE_LOW=1
00>
00> [00:00:05.369,049] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
00> [00:00:05.369,049] <inf> bt_hci_core: HW Variant: nRF52x (0x0002)
00> [00:00:05.369,049] <inf> bt_hci_core: Firmware: Standard Bluetooth controller (0x00)
Version 2.6 Build 0
00> [00:00:05.369,293] <inf> bt_hci_core: No ID address. App must call settings_load()
00> Bluetooth initialised OK
00>
00> E63BE707-2716-0081-1144-7DE53513AB82
00>
00>
00> Mesh initialised OK
00>
00> [00:00:05.371,246] <inf> bt_hci_core: Identity: CD:CD:D3:95:E8:4E (random)
00> [00:00:05.371,246] <inf> bt_hci_core: HCI: version 5.2 (0x0b) revision 0x0000,
manufacturer 0x05f1
00> [00:00:05.371,276] <inf> bt_hci_core: LMP: version 5.2 (0x0b) subver 0xffff
00> Settings loaded
```

```
00>
00> Node has not been provisioned - beaconing
```

The highlighted text reminds us that we have not yet provisioned this node.

Provisioning and Configuration

Since you already learned about the coding required to make it possible to provision a node, we won't repeat that exercise here, and the code for provisioning was already included in the starter code for the light node.

Provision and configure your light node using a provisioning application so that it is a part of the same network as your switch.

Make sure that you

1. Bind the *generic on off server model* to the same application key that the *generic on off client model* was bound to in the switch node.

2. Set the publication state data to use a the same group address as used by the switch, namely 0xC000.

3. Subscribe to the group address 0xC000. This will ensure that your light node responds to *generic on off messages* from your switch node.

After provisioning and configuring the light node, disconnect nRF Mesh from it by selectiung the DISCONNECT button in the UI.

Testing #2

Press buttons 1, 2 and 3 on the switch node while monitoring the light node's console. You should see messages like these, interspersed by other console messages from the stack:

```
00> [00:04:55.654,052] <dbg> bt_mesh_access.bt_mesh_model_recv: app_idx 0x0000 src 0x0002
dst 0xc000
00> [00:04:55.654,083] <dbg> bt_mesh_access.bt_mesh_model_recv: len 4: 82030101
00> [00:04:55.654,083] <dbg> bt_mesh_access.bt_mesh_model_recv: OpCode 0x00008203
00> generic_onoff_set_unack

00> [00:04:58.084,045] <dbg> bt_mesh_access.bt_mesh_model_recv: app_idx 0x0000 src 0x0002
dst 0xc000
00> [00:04:58.084,045] <dbg> bt_mesh_access.bt_mesh_model_recv: len 4: 82030002
00> [00:04:58.084,075] <dbg> bt_mesh_access.bt_mesh_model_recv: OpCode 0x00008203
00> generic_onoff_set_unack

00> [00:05:04.031,005] <dbg> bt_mesh_access.bt_mesh_model_recv: app_idx 0x0000 src 0x0002
dst 0xc000
00> [00:05:04.031,036] <dbg> bt_mesh_access.bt_mesh_model_recv: len 2: 8201
00> [00:05:04.031,036] <dbg> bt_mesh_access.bt_mesh_model_recv: OpCode 0x00008201
00> gen_onoff_get
```

If you see the highlighted console messages then you have verified that mesh messages from the client are being correctly decrypted, recognised as belonging to the generic on off server model and routed to its handler functions.

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 back to the source of the received message as a response whereas the second

does not. In both cases though, the specification states that if a server has a publish address, it is required to publish a status message on a state change (ref Mesh Profile Specification 3.7.6.1.2). If the distinction here is not clear to you, sending a status message to the source of a set message involves setting the dest_addr of the status message to the unicode source address of the set message sender whereas when publishing, we will publish to the Publish Address with which the model has been configured. Typically, this will be a group address.

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)
{
        printk("gen_onoff_set\n");
        set_onoff_state(model, ctx, buf, true);
}

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

And add the *set_onoff_state* function above the two functions you just updated:

```
static void set_onoff_state(struct bt_mesh_model *model, struct bt_mesh_msg_ctx *ctx,
struct net_buf_simple *buf, bool ack)
{
        uint8_t msg_onoff_state = net_buf_simple_pull_u8(buf);
        if (msg_onoff_state == onoff_state) {
                // no state change so nothing to do
                return;
        }
        onoff_state = msg_onoff_state;
        uint8_t tid = net_buf_simple_pull_u8(buf);
        printk("set onoff state: onoff=%u TID=%u\n", onoff_state, tid);
        if (onoff_state == 0)
        {
                thingy_led_off();
        }
        else
        {
                thingy_led_on(rgb_r,rgb_g,rgb_b);
        }

        /*
         * 3.7.7.2 Acknowledged Set
         */
        if (ack) {
                generic_onoff_status(false, onoff_state);
        }

        /*
         * If a server has a publish address, it is required to publish status on a state
change
         * See Mesh Profile Specification 3.7.6.1.2
         */

        if (model->pub->addr != BT_MESH_ADDR_UNASSIGNED) {
                generic_onoff_status(true, onoff_state);
        }

}
```

Explanation

generic_onoff_set_unack simply calls a function *set_onoff_state* to set the onoff state to the value in the received message. It sets the *ack* argument to false to indicate that no acknowledgement is required.

generic_onoff_set calls *set_onoff_state* with the *ack* argument set to true to indicate that this message must be acknowledged with a *generic on off status* message.

set_onoff_state extracts the onoff state field in the received message and checks to see if it contains a different value to the model's current *on off state* or not. If it does not then there is no state change being requested and so nothing to do.

Otherwise, depending on the value of the *onoff_state* field, the function either switches the Thingy LED on or off. If an acknowledgement is required it calls a function called *generic_onoff_status* which we have not yet implemented.

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 Status Message

Under the comment *generic onoff status TX message producer* add the following code.

```
void generic_onoff_status(bool publish, uint8_t on_or_off)
{
    int err;
    struct bt_mesh_model *model = &sig_models[2];
        if (publish && model->pub->addr == BT_MESH_ADDR_UNASSIGNED) {
                printk("No publish address associated with the generic on off server model -
add one with a configuration app like nRF Mesh\n");
                return;
        }

        if (publish) {
                struct net_buf_simple *msg = model->pub->msg;
                net_buf_simple_reset(msg);
                bt_mesh_model_msg_init(msg, BT_MESH_MODEL_OP_GENERIC_ONOFF_STATUS);
                net_buf_simple_add_u8(msg, on_or_off);
                printk("publishing on off status message\n");
                err = bt_mesh_model_publish(model);
                if (err) {
                        printk("bt_mesh_model_publish err %d\n", err);
                }
        } else {
                uint8_t buflen = 7;
                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, on_or_off);
                struct bt_mesh_msg_ctx ctx = {
                                .net_idx = reply_net_idx,
                                .app_idx = reply_app_idx,
                                .addr = reply_addr,
                                .send_ttl = BT_MESH_TTL_DEFAULT,
                };

                printk("sending on off status message\n");
                if (bt_mesh_model_send(model, &ctx, &msg, NULL, NULL))
                {
                        printk("Unable to send generic onoff status message\n");
                }
        }

}
```

Explanation

A *generic on off status* message may either be published to the model publish address that has been configured or it can be sent to an application specified destination address. This function allows the caller to indicate which of the two approaches should be used for transmitting this message type. The value of the *generic on off state* must be supplied for use in the status message.

The function starts with some validation. If the caller has requested that the message be *published* then a *publish address* must have been configured.

If publishing has been requested, a buffer, for containing the message is acquired from the model, it is initialised with the required message opcode and the on off state value added to it. It is then published. If publishing has not been requested, a message buffer is created and populated, a context containing parameters such as the destination address for the message (set to the source address of the message we are replying to) is created and the message is sent.

Generic OnOff Get Message

Update the *generic_onoff_get* function so that 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");
        // logged for interest only
        printk("ctx net_idx=0x%02x\n",ctx->net_idx);
        printk("ctx app_idx=0x%02x\n",ctx->app_idx);
        printk("ctx addr=0x%02x\n",ctx->addr);
        printk("ctx recv_dst=0x%02x\n",ctx->recv_dst);

        reply_addr = ctx->addr;
        reply_net_idx = ctx->net_idx;
        reply_app_idx = ctx->app_idx;
        generic_onoff_status(false, onoff_state);
}
```

Explanation

The source address of the received *generic on off get* message, plus the associated application and network key indexes are saved in variables for use when responding with a status message. The function *generic_onoff_status* is called, indicating that a direct response message is required rather than publication of a status message to all subscribers.

Testing

Build and install your code on the light node. Install the switch code on your other board and ensure buttons 1 and 2 are programmed to send *generic on off set unacknowledged* messages with state values of 1 and 0 respectively. Button 3 should send *generic on off get* messages.

Press button 1. The LED on your light node should come on. Press button 2. The LED should switch off. Press button 3. Because the *generic onoff server model* in the light node has a publish address configured, it will publish a status message to that address on the state change being executed. You should see this status message received in the console of your switch node, assuming you configured its *generic onoff client model* to subscribe to the publish address 0xC000.

Reprogram the switch so that buttons 1 and 2 send *generic on off set* messages i.e. the acknowledged variants. Test again but make sure you also watch the console of the switch node. You should see status messages being received twice from the light when the on off state has been changed. The first will have a destination address (dst) equal to the unicode address of the switch node, as allocated and shown by your provisioning and configuration application. In the example shown here, that dst address is 0x0004. The second will have dst equal to your group address, to which the light will publish and switch has subscribed. This should be 0xC000.

```
publishing set on off state=0x01
onoff set message 1 sent
generic_onoff_status onoff=1
[00:02:15.665,100] <dbg> bt_mesh_access.bt_mesh_model_publish:
[00:02:15.665,130] <dbg> bt_mesh_access.bt_mesh_model_publish: Publish Retransmit Count 1
Interval 50ms
[00:02:15.667,449] <dbg> bt_mesh_access.mod_publish:
[00:02:15.668,792] <dbg> bt_mesh_access.bt_mesh_model_recv: app_idx 0x0000 src 0x00e7 dst
0xc000
[00:02:15.668,823] <dbg> bt_mesh_access.bt_mesh_model_recv: len 4: 82020102
[00:02:15.668,823] <dbg> bt_mesh_access.bt_mesh_model_recv: OpCode 0x00008202
[00:02:15.668,853] <dbg> bt_mesh_access.bt_mesh_model_recv: No OpCode 0x00008202 for elem 0
[00:02:15.677,978] <dbg> bt_mesh_access.bt_mesh_model_recv: app_idx 0x0000 src 0x00e9 dst
0x00e7
[00:02:15.677,978] <dbg> bt_mesh_access.bt_mesh_model_recv: len 3: <log_strdup alloc
failed>
[00:02:15.677,978] <dbg> bt_mesh_access.bt_mesh_model_recv: OpCode 0x00008204
generic_onoff_status onoff=1
[00:02:15.793,914] <dbg> bt_mesh_access.publish_sent: err 0
[00:02:15.799,774] <dbg> bt_mesh_access.bt_mesh_model_recv: app_idx 0x0000 src 0x00e9 dst
0xc000
[00:02:15.799,774] <dbg> bt_mesh_access.bt_mesh_model_recv: len 3: 820401
[00:02:15.799,774] <dbg> bt_mesh_access.bt_mesh_model_recv: OpCode 0x00008204
```

Reinstate the sending of *generic onoff set unacknowledged* messages by your switch node and flash the updated code to your board.

Next

If you're following the recommended sequence, your next step should be to work through the Light HSL Model coding exercises.