Developer Study Guide: An introduction to Bluetooth Mesh Networking

Hands-on Coding Exercise - Generic On Off Client

Release : 2.0.0

Document Version : 2.0.0

Last updated : 16th December 2019

Contents

Revision History

Version	Date	Author	Changes
1.0.0	15th June 2018	Martin Woolley Bluetooth SIG	Initial version
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 nRF52840-DK developer board.

Exercise 3 – Implementing the Generic On/Off Client

Introduction

Your first coding exercise will involve implementing the code required to turn one of your devices into a Bluetooth mesh on/off switch. We'll be using a Nordic nRF52840-DK and make use of two of its four GPIO buttons for this task.
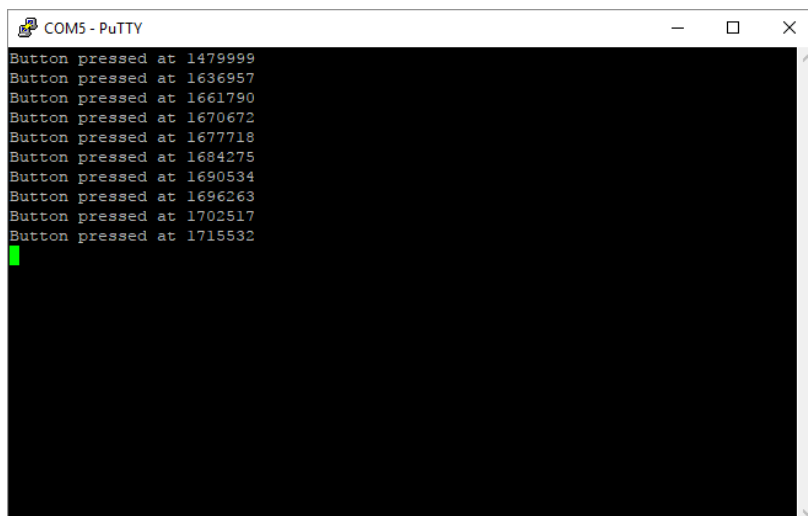
Debugging

As you proceed with your coding and testing, there will be occasions where you want or need to trace execution of the code. This may be because there's a problem you need to solve or purely to reassure yourself that execution is following the path you think it is. Whatever your reasons, here are some tips which will help:

printk and terminal

The *printk* function will write text and variables to a connected serial console provided your device is plugged into your computer via a suitable interface such as USB. For example

```
printk("Button pressed at %d\n", k_cycle_get_32());
```

prints a text message which ends with the current time since the last reset.

Tracing stack messages

The Zephyr framework includes various build flags, some of which are concerned with debugging and which enable or disable the output of debug messages from various layers of the stack. Flags such as the ones shown below, should be included in the project's prj.conf file.

```
CONFIG_BT_DEBUG_LOG=y
CONFIG_BT_MESH_DEBUG=y
CONFIG_BT_MESH_DEBUG_MODEL=y
CONFIG_BT_MESH_DEBUG_ACCESS=y
CONFIG_BT_MESH_DEBUG_MODEL=y
CONFIG_BT_MESH_DEBUG_TRANS=n
CONFIG_BT_MESH_DEBUG_NET=n
CONFIG_BT_MESH_DEBUG_CRYPTO=n
```

and after you next build and install the resultant hex file, you'll see entries like the following ones appearing in the serial console connected to your device:

```
[bt] [DBG] model_send: (0x20001378) net_idx 0x0000 app_idx 0x0000 dst 0xc000
[bt] [DBG] model_send: (0x20001378) len 2: 8201
onoff get message sent
[bt] [DBG] bt_mesh_model_recv: (0x20001294) app_idx 0x0000 src 0x0007 dst 0x0001
[bt] [DBG] bt_mesh_model_recv: (0x20001294) len 4: 82040000
[bt] [DBG] bt_mesh_model_recv: (0x20001294) OpCode 0x00008204
[bt] [DBG] bt_mesh_model_recv: (0x20001294) No OpCode 0x00008204 for elem 0
```

Testing

You won't be able to fully test your code until you've implemented the generic on/off server model in the next exercise, so initially you will simply review debug messages in the console to gain some degree of confirmation that your code is functioning the way it should.

Project Set Up

Create the following directories for your project:

```
switch/
    build/
    src/
```

We'll refer to the root directory of the mesh developer study guide as $MDG from now on. Copy all of the files and the src directory from $MDG\code\start_state\Switch\ to your project's root directory.

Your project directory should contain the following files:

Name	Date modified	Type	Size
build	17/12/2019 11:01	File folder	
src	17/12/2019 10:25	File folder	
CMakeLists.txt	13/12/2019 12:58	Text Document	1 KB
prj.conf	13/12/2019 12:58	CONF File	2 KB
sample.yaml	13/12/2019 12:58	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 switch node in this exercise.

We used a nRF52840-DK developer board for the switch node. To build the starter code for this board, the following Zephyr west command must be executed from within your project's root directory:

```
west build -b nRF52840_PCA10056
```

The starter code should compile and link. Output should include lines similar to this:

```
C:\mdsg\switch>west build -b nRF52840_PCA10056
source directory: C:\mdsg\switch
build directory: C:\mdsg\switch\build
BOARD: nRF52840_PCA10056
Zephyr version: 1.14.1
```

```
-- Found PythonInterp: C:/Python37/python.exe (found suitable version "3.7.3", minimum
required is "3.4")
-- Selected BOARD nRF52840_PCA10056
-- Found west: C:/Python37/Scripts/west.exe (found suitable version "0.6.2", minimum
required is "0.5.6")
-- Loading
C:/workspaces/zephyr_source/zephyr/boards/arm/nrf52840_pca10056/nRF52840_PCA10056.dts as
base
-- Overlaying c:/workspaces/zephyr_source/zephyr/dts/common/common.dts
Parsing Kconfig tree in c:/workspaces/zephyr_source/zephyr/Kconfig
Loading
C:/workspaces/zephyr_source/zephyr/boards/arm/nrf52840_pca10056/nRF52840_PCA10056_defconfig
as base
Merging C:/mdsg/switch/prj.conf
Configuration written to 'C:/mdsg/switch/build/zephyr/.config'
-- Cache files will be written to: C:\Users\mwoolley\AppData\Local/.cache/zephyr
-- 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
-- Performing Test toolchain_is_ok
-- Performing Test toolchain_is_ok - Success
Including module: tinycbor in path: c:\workspaces\zephyr_source\modules\lib\tinycbor
-- Configuring done
-- Generating done
-- Build files have been written to: C:/mdsg/switch/build
[130/187] Building C object CMakeFiles/app.dir/src/main.c.obj
../src/main.c: In function 'main':
../src/main.c:320:6: warning: unused variable 'err' [-Wunused-variable]
  int err;
      ^~~
At top level:
../src/main.c:76:43: warning: 'health_srv_cb' defined but not used [-Wunused-const-
variable=]
 static const struct bt_mesh_health_srv_cb health_srv_cb = {
                                           ^~~~~~~~~~~~~
../src/main.c:62:22: warning: 'dev_uuid' defined but not used [-Wunused-const-variable=]
 static const uint8_t dev_uuid[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08,0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x00 };
                      ^~~~~~~~
[182/187] Linking C executable zephyr\zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:      141658 B         1 MB     13.51%
            SRAM:       21300 B       256 KB      8.13%
        IDT_LIST:         120 B         2 KB      5.86%
[187/187] Linking C executable zephyr\zephyr.elf

C:\mdsg\switch>
```

The warnings shown here are to be expected. We'll resolve them later. But 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.

Tracing Mesh Messages

It is recommended that you have all your Zephyr projects enable at least some of the
CONFIG_BT_MESH_DEBUG_XXXX settings in prj.conf and while developing and testing, always have
your device connected to your computer over USB. Connect your favourite terminal program (e.g.
Putty) to it over the appropriate serial port. This will allow you to view console output from the
Zephyr framework and produced from your own code using the *printk* function.

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*. We need to define the composition of our
node in code. We'll start by dealing with the required foundation models (configuration server and
health server) and the generic on off client model only. We'll add the light HSL client model later on.

Configuration Server Models

A node must support the Configuration Server Model. 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_ENABLED,
          .default_ttl = 7,
          /* 3 transmissions with 20ms interval */
          .net_transmit = BT_MESH_TRANSMIT(2, 20),
};
```

Explanation

This struct specifies some of the special state values which the configuration server model contains. In our case, it indicates that our node will be able to act as a proxy, that the default TTL for any messages sent is 7 and that at the network layer, a copy of every message transmitted will be sent a total of 3 times, at 20ms intervals at most. The exact timing of these transmissions is dependent upon the bearer layer and the capabilities of the underlying advertising capability, so this is an approximation.

The reason we've enabled the proxy feature is to allow provisioning over GATT, which is supported by nRF Mesh and similar applications.

We'll use this struct later on, as we complete the definition of our node's composition.

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.

Mesh Messages

Generic OnOff Message Types

Under the comment *generic on off client - message types defined by this model*, 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 client 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 for each message opcode, a function which will handle messages of that type.

Add this code under the constants you just defined.

```
static const struct bt_mesh_model_op gen_onoff_cli_op[] = {
    {BT_MESH_MODEL_OP_GENERIC_ONOFF_STATUS, 1, generic_onoff_status},
    BT_MESH_MODEL_OP_END,
};
```

Explanation

This array of *bt_mesh_model_op* types contains a single significant member item, which specifies the opcode for the Generic OnOff Status message, the only type of message which our Generic OnOff Client must be able to receive and process. It specifies that access message payloads must be at least 1 octet long and indicates that a function called generic_onoff_status must handle all such messages received.

BT_MESH_MODEL_END indicates the end of the definition.

Skeleton Generic On Off Status Message Handler Function

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

Under the comment *generic on off client - handler functions for this model's RX* messages add the following:

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

Explanation

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

List the Models and Link To Message Handler Function Definitions

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

```
BT_MESH_MODEL_PUB_DEFINE(gen_onoff_cli, NULL, 2);

static struct bt_mesh_model sig_models[] = {
    BT_MESH_MODEL_CFG_SRV(&cfg_srv),
    BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
    BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_ONOFF_CLI, gen_onoff_cli_op, & gen_onoff_cli,
&onoff[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 server model and the health server model. We've used a general purpose model definition macro to define the generic onoff client model. As you can see, these definitions reference the other definitions we prepared earlier on. We've also defined a model publication context for the generic on off client model so that we can set and use various message publication states.

Define Elements and Contained Models

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 = {
    .cid = 0xFFFF,
    .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 hiererchical definition of the node and its composition, starting with the elements directly owned by the node. This struct also specifies a Company ID (CID) of 0xFFFF. This is a special value which must only be used during testing. For products to be released, a value must be assigned for your company by the Bluetooth SIG. See https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers/

Checkpoint

Build your code with the *west build -b nRF52840_PCA10056* command from within your project's root directory. It should compile and link now but will generate a number of warnings relating to items we've defined but not yet used.

```
C:\mdsg\switch>west build -b nRF52840_PCA10056
source directory: C:\mdsg\switch
build directory: C:\mdsg\switch\build
BOARD: nRF52840_PCA10056
[1/7] Building C object CMakeFiles/app.dir/src/main.c.obj
../src/main.c: In function 'main':
../src/main.c:361:6: warning: unused variable 'err' [-Wunused-variable]
  int err;
      ^~~
At top level:
../src/main.c:185:34: warning: 'comp' defined but not used [-Wunused-const-variable=]
 static const struct bt_mesh_comp comp = {
                                  ^~~~
../src/main.c:76:43: warning: 'health_srv_cb' defined but not used [-Wunused-const-
variable=]
```

```
  static const struct bt_mesh_health_srv_cb health_srv_cb = {
                                             ^~~~~~~~~~~~~~
../src/main.c:62:22: warning: 'dev_uuid' defined but not used [-Wunused-const-variable=]
  static const uint8_t dev_uuid[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08,0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x00 };
                       ^~~~~~~~
[3/7] Linking C executable zephyr\zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:       141658 B         1 MB     13.51%
            SRAM:        21300 B       256 KB      8.13%
        IDT_LIST:          120 B         2 KB      5.86%
[7/7] Linking C executable zephyr\zephyr.elf
```

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.

Bluetooth Stack Initialisation

Your next job is to initialise the Bluetooth and Bluetooth mesh stacks. After that we'll provision and configure using the smartphone application nRF Mesh.

Update your main function so that it looks like this:

```
void main(void)
{
        int err;
        printk("switch\n");

        onoff_tid = 0;
        hsl_tid = 0;

        configureButtons();

        configureLED();

        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 any of the four buttons on the board 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.

We also call a function configureLED() whose source you will also find in the starter code. This function sets up GPIO to allow the board's built-in LED to be switched on or off and switches it off to begin with.

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

Provisioning Configuration

To initialise the mesh stack, we need to provide a struct containing information relating to how we want to handle provisioning and a struct which defines our node composition. We already defined the latter (see variable *comp* of type *bt_mesh_comp*).

The Zephyr API defines a type, *bt_mesh_prov* which allows a struct containing various provisioning related properties to be defined. Add the following under the *provisioning properties and capabilities* comment:

```
static const struct bt_mesh_prov prov = {
        .uuid = dev_uuid,
        .output_size = 4,
        .output_actions = BT_MESH_DISPLAY_NUMBER,
        .output_number = provisioning_output_pin,
        .complete = provisioning_complete,
        .reset = provisioning_reset,
};
```

Explanation

When a device is available for provisioning, it will advertise *unprovisioned device beacons*. These special advertising packets include a *device UUID* which identifies the device. The value of device UUID is set by the manufacturer in the factory. For convenience, a suitable value has been hard-coded in the starter code and this is referenced in the struct's *uuid* property.

There are a number of ways in which an unprovisioned device may authenticate itself to the user during the provisioning process. We want to offer the user the option to have the device authenticate itself by displaying a 4 digit PIN. The property *output_actions* indicates that this is the case, with *output_size* specifying the required PIN length. The function specified as the *output_number* property will be called with a random PIN generated by the stack for display to the user.

The *complete* and *reset* properties name functions to be called when provisioning has either completed or been abandoned, respectively.

We'll implement these functions soon.

Mesh Initialisation, Settings restore and Provisioning Status Check

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");

        // prov is a bt_mesh_prov struct and is declared in provisioning.c
        err = bt_mesh_init(&prov, &comp);
        if (err)
        {
            printk("bt_mesh_init failed with err %d\n", err);
            return;
        }

        printk("Mesh initialised OK\n");

        if (IS_ENABLED(CONFIG_SETTINGS)) {
```

```
        settings_load();
        printk("Settings loaded\n");
    }

    if (!bt_mesh_is_provisioned()) {
        printk("Node has not been provisioned - beaconing\n");
        bt_mesh_prov_enable(BT_MESH_PROV_ADV | BT_MESH_PROV_GATT);
    } else {
        printk("Node has already been provisioned\n");
    }

}
```

Explanation

bt_ready is called after the Bluetooth stack has been initialised and so the first task is to verify that this was successful by checking the value of the *err* parameter.

We then initialise the mesh stack by calling *bt_mesh_init* with our provisioning and node composition structs as arguments.

Next, we call *settings_load()*, which restores persisted data including mesh stack variables such as the SEQ field (so we can continue from the last allocated value) and provisioning data such as the network key and application key, if the device has already been provisioned.

We check whether or not the node has already been provisioned, using the function *bt_mesh_is_provisioned()* and if it has not, start broadcasting unprovisioned device beacons by calling *bt_mesh_prov_enable()*. The arguments BT_MESH_PROV_ADV | BT_MESH_PROV_GATT indicate that provisioning may be performed using either the advertising bearer or the GATT bearer.

Provisioning Behaviours

We now need to make some changes which will allow our device to be provisioned in the required way.

Provisioning Callback Functions

Add the following functions above the bt_mesh_prov struct definition.

```
static int provisioning_output_pin(bt_mesh_output_action_t action, u32_t number) {
        printk("OOB Number: %04d\n", number);
        return 0;
}

static void provisioning_complete(u16_t net_idx, u16_t addr) {
    printk("Provisioning completed\n");
}

static void provisioning_reset(void)
{
        bt_mesh_prov_enable(BT_MESH_PROV_ADV | BT_MESH_PROV_GATT);
}
```

Explanation

The function *provisioning_output_pin* simply outputs the PIN generated by the stack to the console. Other devices might have more sophisticated UI options. The serial console is good enough for our purposes.

provisioning_complete just outputs a message to the console to indicate that provisioning has finished.

reset is called when provisioning was not completed (e.g. abandoned by the user). In this function, we restart the advertising of unprovisioned device beacons.

Attention

The health server model allows a device to do something which will bring it to the attention of a person for some reason (e.g. to indicate a fault). This feature may also be used during provisioning in the *Provisioning Invite* step of the provisioning procedure and we'll use it to have the board light its LED so we can see the device that is being communicated with during provisioning.

Update the health_srv_cb struct so that it looks like this:

```
static const struct bt_mesh_health_srv_cb health_srv_cb = {
        .attn_on = attention_on,
        .attn_off = attention_off,
};
```

And update *health_srv* to reference the *health_srv_cb* struct.

```
static struct bt_mesh_health_srv health_srv = {
        .cb = &health_srv_cb,
};
```

In the health server section of our code, above the health_srv_cb struct, add these functions:

```
static void attention_on(struct bt_mesh_model *model)
{
        printk("attention_on()\n");
        ledOn();
}

static void attention_off(struct bt_mesh_model *model)
{
        printk("attention_off()\n");
        ledOff();
}
```

Explanation

These two functions are called automatically, during provisioning and write a suitable message to the console and either light or switch off the board's LED, as appropriate.

Checkpoint

Build your code and flash it to your board over USB with the *west flash* command. It should compile and link without warnings or errors and flash to your board.

```
C:\mdsg\Switch>west flash
[3/7] Linking C executable zephyr\zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:      190046 B         1 MB     18.12%
            SRAM:       24040 B       256 KB      9.17%
        IDT_LIST:         120 B         2 KB      5.86%
[7/7] Linking C executable zephyr\zephyr.elf
Using runner: nrfjprog
Flashing file: C:/mdsg/Switch/build/zephyr/zephyr.hex
Parsing hex file.
Erasing page at address 0x0.
Erasing page at address 0x1000.
```

```
Erasing page at address 0x2000.
Erasing page at address 0x3000.
Erasing page at address 0x4000.
Erasing page at address 0x5000.
Erasing page at address 0x6000.
Erasing page at address 0x7000.
Erasing page at address 0x8000.
Erasing page at address 0x9000.
Erasing page at address 0xA000.
Erasing page at address 0xB000.
Erasing page at address 0xC000.
Erasing page at address 0xD000.
Erasing page at address 0xE000.
Erasing page at address 0xF000.
Erasing page at address 0x10000.
Erasing page at address 0x11000.
Erasing page at address 0x12000.
Erasing page at address 0x13000.
Erasing page at address 0x14000.
Erasing page at address 0x15000.
Erasing page at address 0x16000.
Erasing page at address 0x17000.
Erasing page at address 0x18000.
Erasing page at address 0x19000.
Erasing page at address 0x1A000.
Erasing page at address 0x1B000.
Erasing page at address 0x1C000.
Erasing page at address 0x1D000.
Erasing page at address 0x1E000.
Erasing page at address 0x1F000.
Erasing page at address 0x20000.
Erasing page at address 0x21000.
Erasing page at address 0x22000.
Erasing page at address 0x23000.
Erasing page at address 0x24000.
Erasing page at address 0x25000.
Erasing page at address 0x26000.
Erasing page at address 0x27000.
Erasing page at address 0x28000.
Erasing page at address 0x29000.
Erasing page at address 0x2A000.
Erasing page at address 0x2B000.
Erasing page at address 0x2C000.
Erasing page at address 0x2D000.
Erasing page at address 0x2E000.
Applying system reset.
Checking that the area to write is not protected.
Programming device.
Enabling pin reset.
Applying pin reset.


Board with serial number 683470605 flashed successfully.
```

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.

RX Messages

Our switch implements the configuration server model, the health server model and the generic onoff client model. The mesh models specification defines the rules for supporting associated message types, in terms of being mandatory, optional or conditional. For example, here's an extract from the specification showing the rules regarding the generic onoff client and message support:

Element	SIG Model ID	Procedure	Messages	Rx	Tx
Main	0x1001	Generic OnOff	Generic OnOff Get		O
			Generic OnOff Set		O
			Generic OnOff Set Unacknowledged		O
			Generic OnOff Status	C.1	

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

Table 3.116: Generic OnOff Client elements and messages

As you'll learn when we get to the implementation of the light node, the generic onoff server mandates support for all four of the messages defined for the generic onoff client. So to allow us to test the generic onoff server functionality of our light, we'll implement all of the client messages.

Note that 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.

We'll start with the sole RX message that the switch must support, the generic onoff 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 or as responses to acknowledged SET messages but can also be sent at any time by the server.

Generic OnOff Status

We've registered a function to handle generic onoff status messages and added a skeleton implementation of that function. There's not much left to do in our case. Modify the generic_onoff_status function so that it logs the received *generic on off status* value.

```
static void generic_onoff_status(struct bt_mesh_model *model,struct bt_mesh_msg_ctx
*ctx,struct net_buf_simple *buf)
{
        u8_t onoff_state = net_buf_simple_pull_u8(buf);
        printk("generic_onoff_status onoff=%d\n",onoff_state);
}
```

Explanation

Earlier in the exercise, you defined an array called gen_onoff_cli_op which maps message opcodes to functions, including in this case, the opcode for the generic onoff status message, which you associated with the generic_onoff_status function:

```
static const struct bt_mesh_model_op generic_onoff_cli_op[] = {
{BT_MESH_MODEL_OP_GENERIC_ONOFF_STATUS, 1, generic_onoff_status},
        BT_MESH_MODEL_OP_END,
};
```

You also associated this opcode/function mapping with the generic onoff 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_HEALTH_SRV(&health_srv, &health_pub),
        BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_ONOFF_CLI, gen_onoff_cli_op, NULL, &onoff[0]),
};
```

This allows received messages with the generic onoff status opcode to be routed to the generic_onoff_status function for processing.

In the handler function, we use the one of the [Zephyr buffer processing APIs,](#) net_buf_simple_pull_u8 to extract the single octet *generic onoff state* value which we include in our console message.

TX Messages

We'll now implement the three TX messages. We'll trigger sending the TX messages using two buttons. For normal use we'll send generic onoff set unacknowledged messages with a value of 1 if button 1 is pressed and a value of 0 if button 2 is pressed. We'll temporarily link the other message types to the two buttons by commenting code out so that we can test them.

Generic OnOff Get

Add the following function to the *Generic OnOff Client - TX message producer functions* section:

```
int genericOnOffGet()
{
    printk("genericOnOffGet\n");
    int err;
    struct bt_mesh_model *model = &sig_models[2];
    if (model->pub->addr == BT_MESH_ADDR_UNASSIGNED) {
        printk("No publish address associated with the generic on off client model - add one
with a configuration app like nRF Mesh\n");
        return -1;
    }
    struct net_buf_simple *msg = model->pub->msg;

    bt_mesh_model_msg_init(msg,BT_MESH_MODEL_OP_GENERIC_ONOFF_GET);
    printk("publishing get on off message\n");
    err = bt_mesh_model_publish(model);
    if (err) {
        printk("bt_mesh_model_publish err %d\n", err);
    }
    return err;
}
```

Explanation

Zephyr API provide two ways of sending mesh messages.

Publishing involves transmitting a message to an address that has been specified as the *publish address* for the relevant model in this element using a configuration client. See section 4.2.2 of the mesh profile specification for more information on model publication and associated states.

Sending involves application code explicitly supplying parameters such as the destination address for the message.

In most cases, *publishing* will be used.

In our function, we first check that a model publish address has been configured. If there has not, we ignore the call and return immediately.

Using the Zephyr APIs, we acquire a buffer for our message from the generic on off client model and prepare it for use by calling *bt_mesh_model_msg_init*. In the case of other message types, we might have needed to set some message argument values in the buffer but in this case we do not need to since *generic on off get* has no arguments.

bt_mesh_model_publish(model) causes our message to be published.

The Generic OnOff Set message types

The code required to send a *generic onoff set unacknowledged* message only differs from the code used when sending a *generic onoff 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 OnOff Client - TX message producer functions*:

```
int sendGenOnOffSet(u8_t on_or_off, u16_t message_type)
{
    int err;
    struct bt_mesh_model *model = &sig_models[2];
    if (model->pub->addr == BT_MESH_ADDR_UNASSIGNED) {
        printk("No publish address associated with the generic on off client model - add one
with a configuration app like nRF Mesh\n");
        return -1;
    }
    struct net_buf_simple *msg = model->pub->msg;
    bt_mesh_model_msg_init(msg,        message_type);
    net_buf_simple_add_u8(msg, on_or_off);
    net_buf_simple_add_u8(msg, onoff_tid);
    onoff_tid++;
    printk("publishing set on off state=0x%02x\n",on_or_off);
    err = bt_mesh_model_publish(model);
    if (err) {
        printk("bt_mesh_model_publish err %d\n", err);
    }
    return err;
}
```

Explanation

The function takes two parameters, the first of which is the onoff state value which should be 0 or a 1. The second is an opcode which will indicate which of *generic onoff set* or *generic onoff set unacknowledged* we wish to send.

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

Field	Size (octets)	Notes
OnOff	1	The target value of the Generic OnOff 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.

Table 3.35: Generic OnOff Set message parameters

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

Once again, we check for a publish address for this model, acquire a message buffer and initialise it.

We add the onoff state and TID (Transaction Identifier) values to the message buffer using the Zephyr function *net_buf_simple_add_u8*, increment the TID in readiness for publishinging our next message and then publish the message using the Zephyr *bt_mesh_model_publish* function.

Generic OnOff Set

Add the following function, which uses our common sendGenOnOffSet function to send an acknowledged *generic onoff set* message:

```
void genericOnOffSet(u8_t on_or_off)
{
        if (sendGenOnOffSet(on_or_off, BT_MESH_MODEL_OP_GENERIC_ONOFF_SET))
        {
                printk("Unable to send generic onoff set message\n");
        } else {
                printk("onoff set message %d sent\n",on_or_off);
        }
}
```

Generic OnOff Set Unacknowledged

Add the following function to send a *generic onoff set unacknowledged* message:

```
void genericOnOffSetUnAck(u8_t on_or_off)
{
        if (sendGenOnOffSet(on_or_off, BT_MESH_MODEL_OP_GENERIC_ONOFF_SET_UNACK))
        {
                printk("Unable to send generic onoff set unack message\n");
        } else {
            printk("onoff set unack message %d sent\n",on_or_off);
        }
}
```

Build and flash your updated code with *west flash*.

Provisioning and Configuration

Your device is now in a suitable state to be provisioned and configured. We'll assume you are using the nRF Mesh application for smartphones in this section.
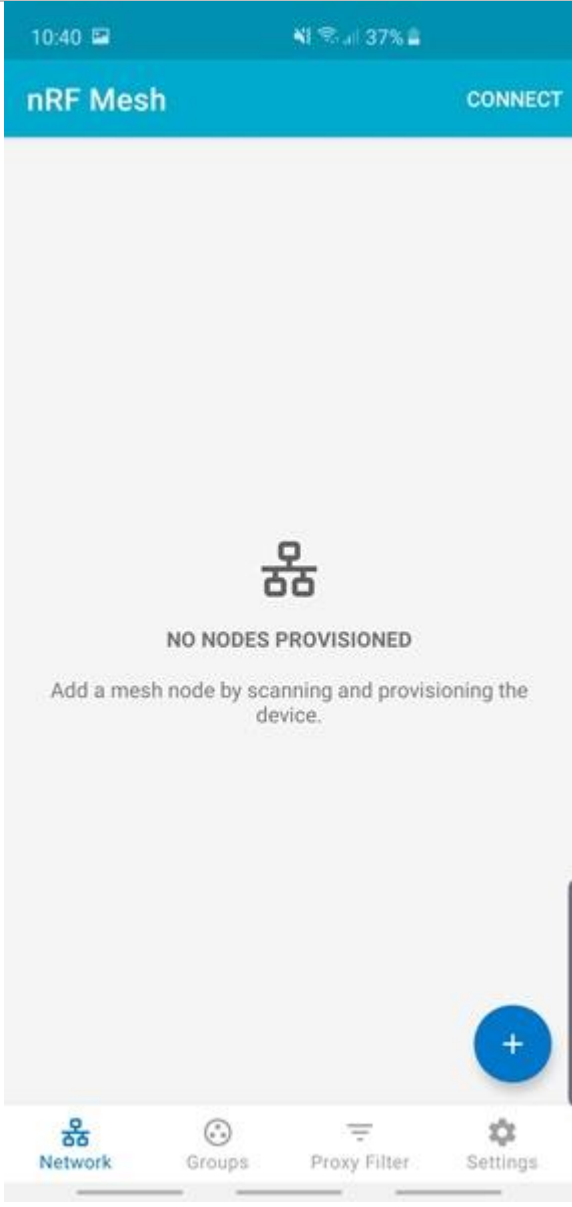
Provisioning

Connect your developer board to your computer using a USB cable and then connect a serial terminal such as Putty to it.
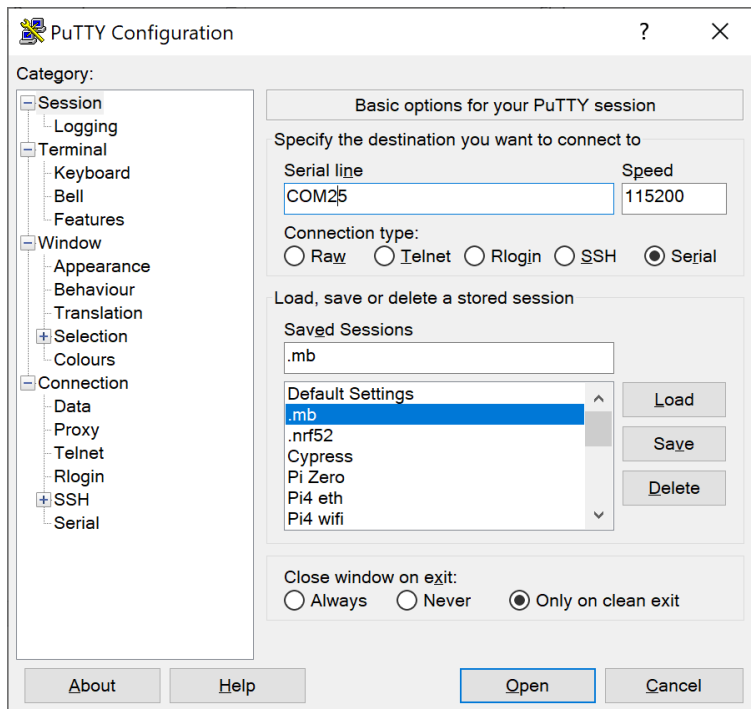
Launch nRF Mesh on your smartphone. From the Network screen, select + ADD NODE. This will start the app scanning for devices that are advertising as unprovisioned mesh devices. You should see your device discovered with the name SWITCH (see CONFIG_BT_DEVICE_NAME="SWITCH" in prj.conf).

The following sequence of screenshots illustrate the steps involved in provisioning the switch node:

Initially you should have no devices listed in the Network screen.

Select + to start scanning for unprovisioned devices.
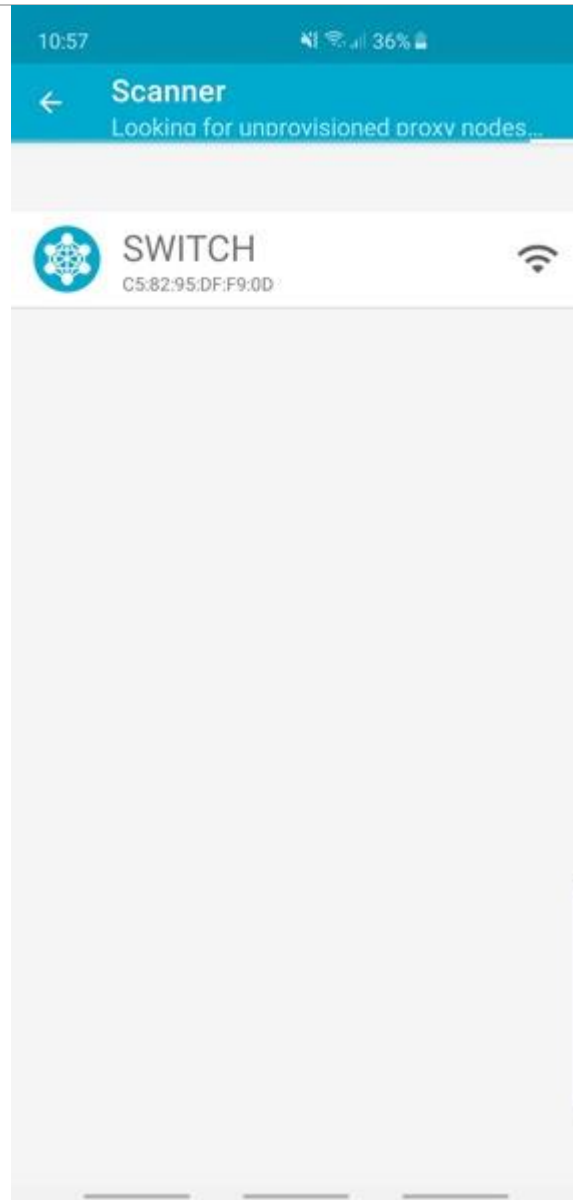
Here, the Switch node has been found. Note that no information other than its current Bluetooth device address is displayed.

Select the SWITCH node in the UI.
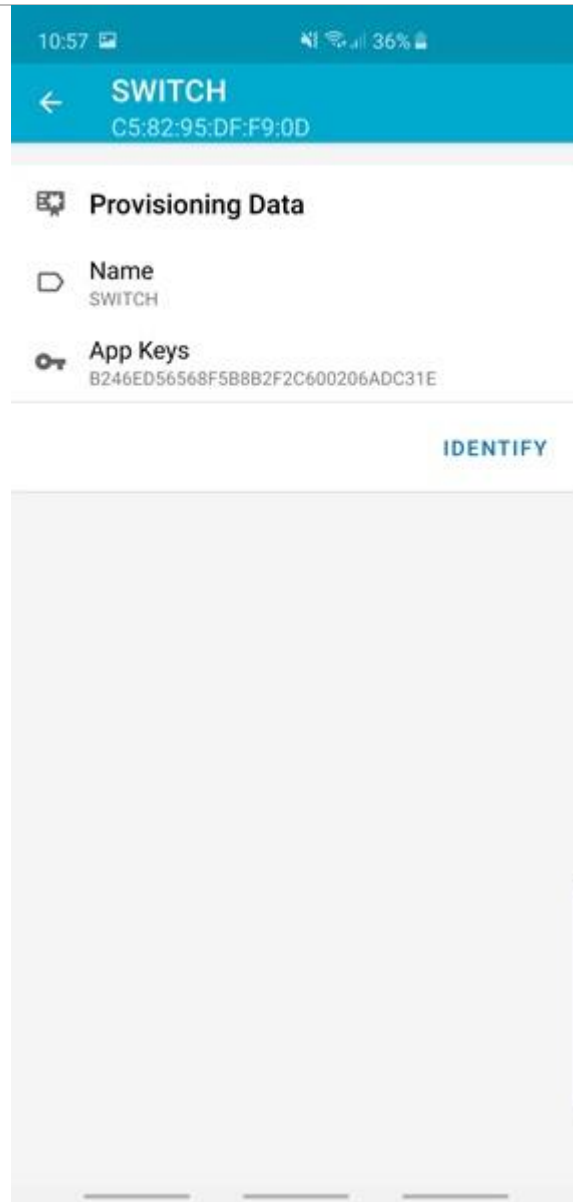
Select IDENTIFY and watch your developer board. This will trigger a call to the attention_on() function and the LED on your board should light.
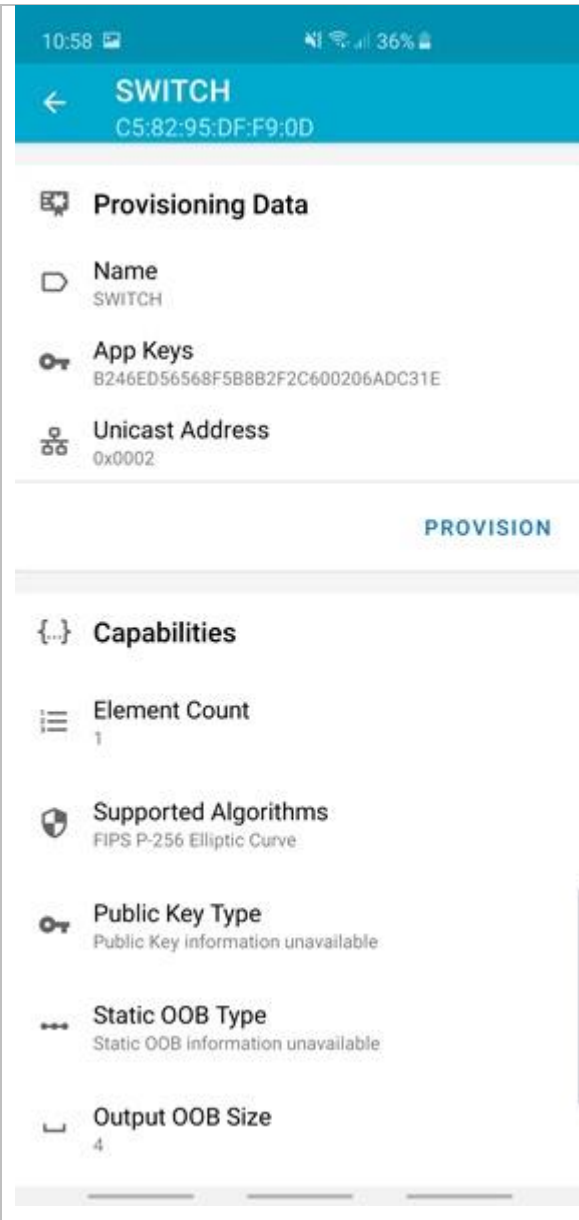
Now select PROVISION.
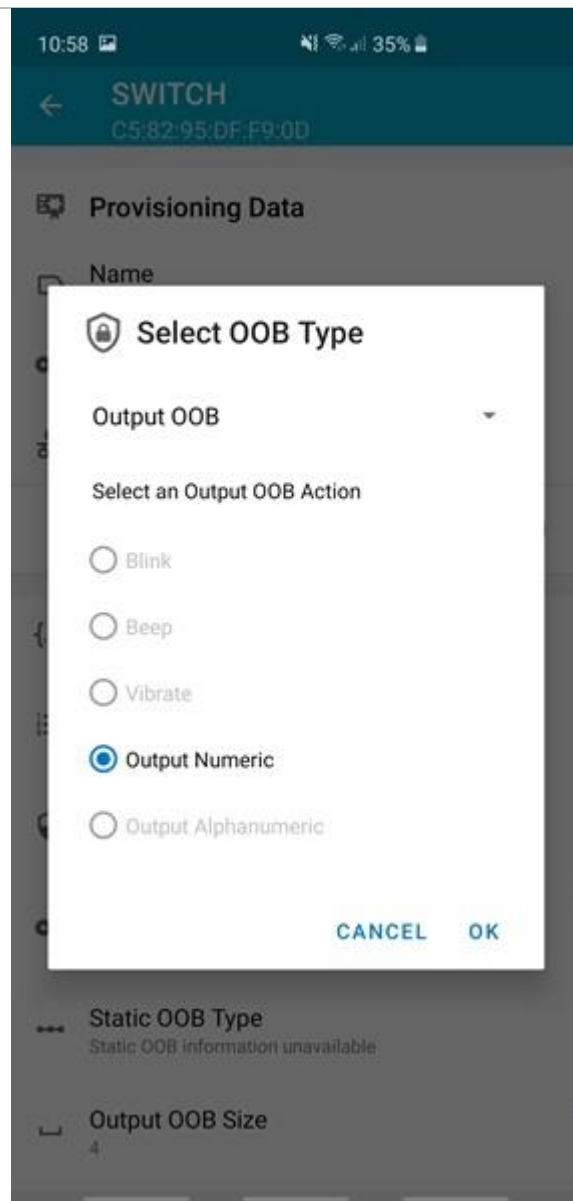
The next dialogue allow you to decide whether or not you want the device to authenticate itself. Select Output OOB to indicate that the device should generate a PIN with which to perform authentication and then select OK.

Your device will now proceed to generate a PIN and your provisioning_output_pin() function will output it to the console.

Note the messages from the attention_on() and attention_off() functions. The LED on your developer board should light when attention_on() is called and be switched off when attention_off() is called.
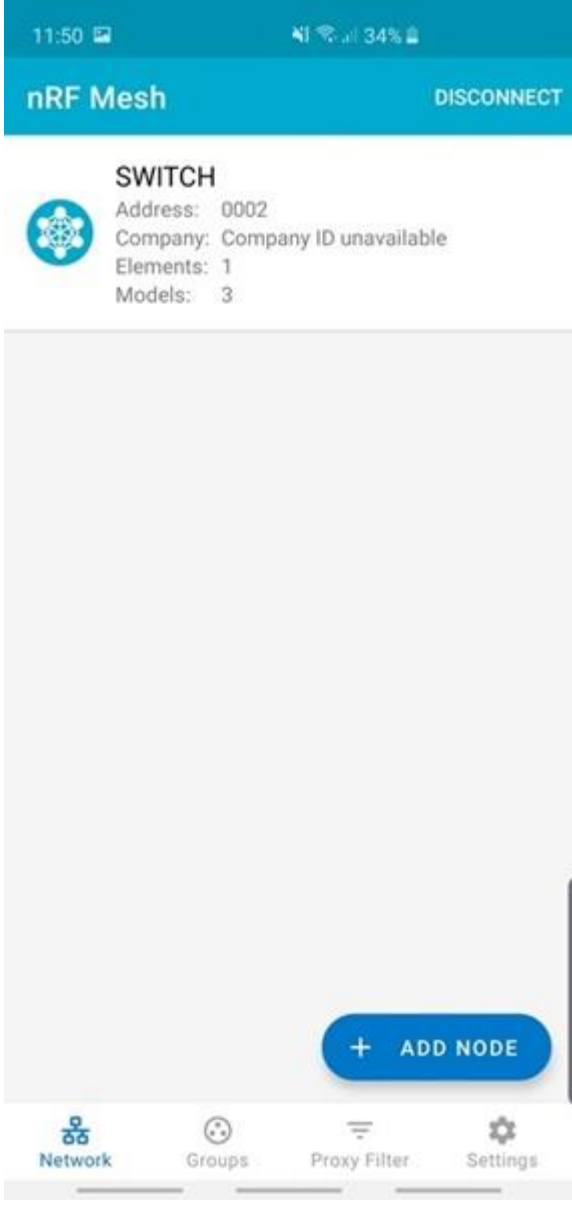
The OOB number is the PIN which you should type into nRF Mesh in the dialogue shown next.
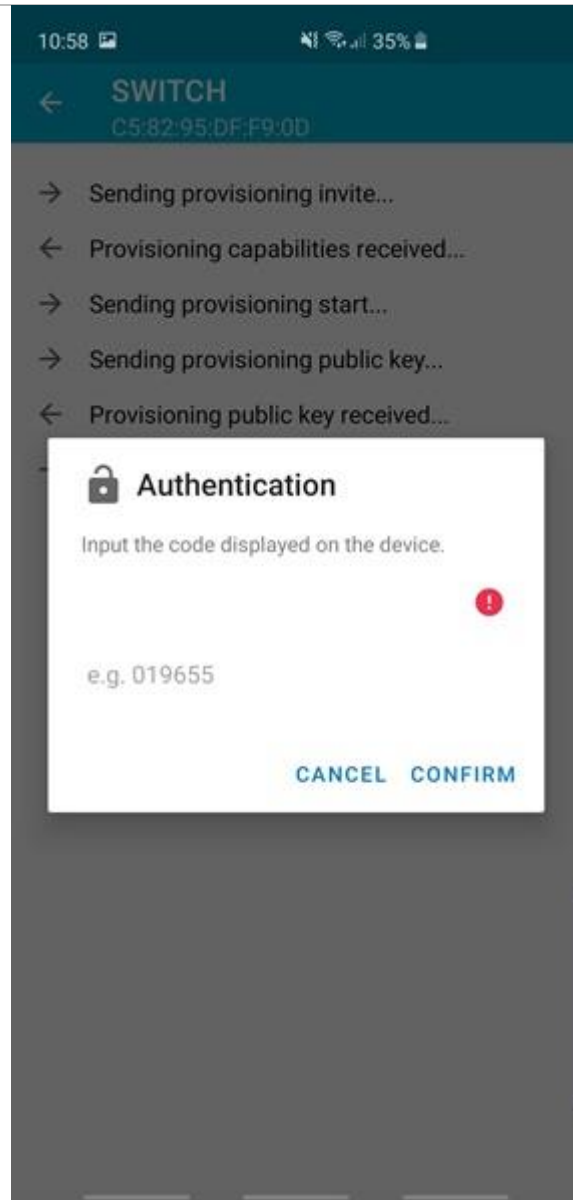
Touch this dialogue's CONFIRM button close to the example PIN text "e.g. 019655" and enter the OOB number which was output to the console.

10:59

SWITCH
C5:82:95:DF:F9:0D

→ Sending provisioning invite...

← Provisioning capabilities received...

→ Sending provisioning start...

→ Sending provisioning public key...

← Provisioning public key received...

→ Waiting for user authentication input...

✓✓ Configuration Complete

Mesh node has been successfully
configured.

OK

→ Sending provisioning data...

← Provisioning complete received...

→ Sending composition data get...

→ Sending block acknowledgements

← Composition data status received...

→ Sending block acknowledgements

→ Sending default TLL get...

→ Default TTL status received...

Don't be fooled by the message in this dialogue! Provisioning has now completed but there are still some configuration steps we need to perform.

SWITCH
Address: 0002
Company: Company ID unavailable
Elements: 1
Models: 3

+ ADD NODE

Network   Groups   Proxy Filter   Settings

Your provisioned device is now listed on the Network screen and it includes a number of details such as the device address that was allocated to it, the fact that it has a single addressable element and that this element, as expected, currently contains three models. The Company name is a consequence of the CID (Company ID) field having been set to the testing value of 0xFFFF.

Configuration

If necessary, select CONNECT to scan for your newly provisioned device. When it appears on screen, select it to connect to it. Note that it is its capability to act as a mesh proxy that we are now using.

Expand the list of models associated with the element. Go into the Generic OnOff Client model. Select BIND KEY and choose Application Key 1. This binds application key 1 to the *generic on off client* model so that it is used for the encryption and decryption of parts of mesh PDUs relating to the upper layers of the stack in mesh messages belonging to this model.

Select SET PUBLICATION. Select Publish Address and Select type Groups and opt to create group address 0xC000. Give it a suitable name such as Lights. Now, whenever this model publishes a message, it will use a destination address of 0xC000. We will subscribe our light node to this address.

In the Publish Retransmission section, set Retransmit Count to zero so that it is marked as *disabled*.

Apply the changes you have made so far. An issue exists with v2.1.4 of the nRF Mesh application (used when this study guide was created) whereby the Publish Retransmission state values need to be applied twice for them to change. To be sure this has worked, select SET PUBLICATION again and make sure Retransmit Count is set to zero.

Select SUBSCRIBE. Select the group address 0xC000.
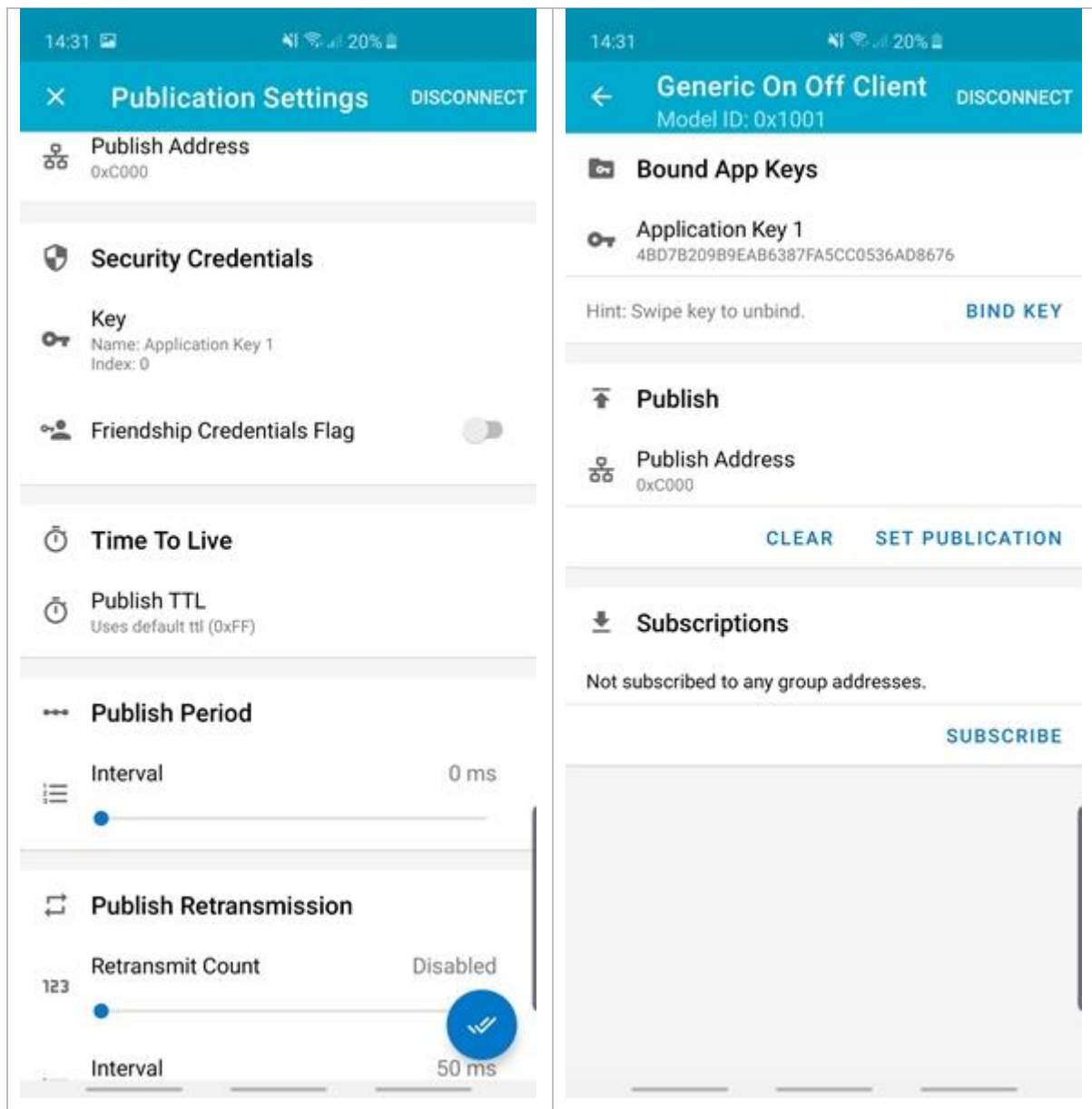
Starting Again

If you need to start again with provisioning and configuration, go into the Settings screen of nRF Mesh and select Reset from the menu. This clears your provisioning database on your smartphone but does not reset your device. To clear the provisioning and configuration data on your mesh device, assuming you are using the board types suggested in the Coding Exercises Introduction, run the command *nrfjprog -e* with your board plugged in over USB. You will need to install the Nordic Command Line Tools to have nrfjprog available.

Testing the On/Off Switch

On the assumption that you do not currently have a device which implements the *generic on off server* model available to you, we are limited regarding the testing we can currently do. At this stage therefore, we will constrain ourselves to triggering various types of message being published whenever button 1 or 2 are pressed and checking for reasonable looking entries in the output to the console. When you have implemented the *generic on off server model* for your light node, we'll be able to perform more thorough testing.

on/off test 1

Button 1 : send a generic onoff set acknowledged (ON)

Button 2 : send a generic onoff set acknowledged (ON)

Button 3 : send a generic onoff get

Adjust the statements in your button1_work_handler, button2_work_handler and button3_work_handler functions so that they look like this:
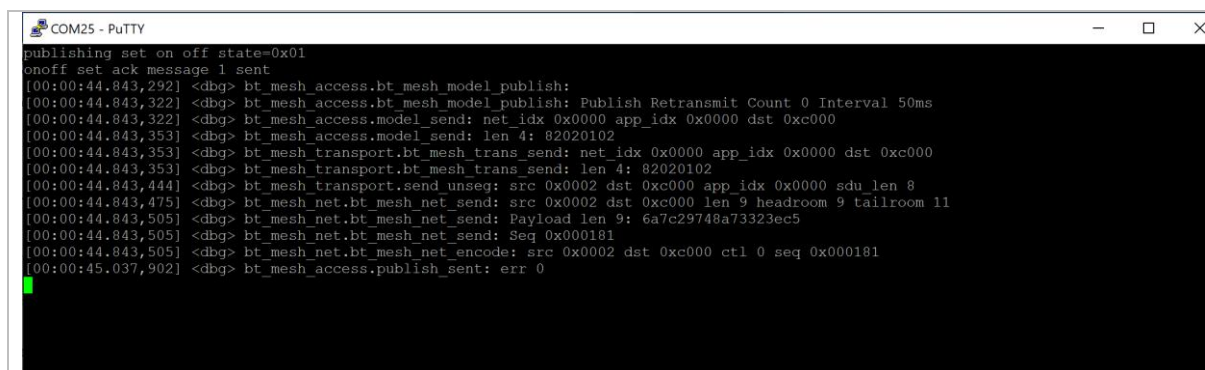
```
void button1_work_handler(struct k_work *work)
{
    genericOnOffSetUnAck(onoff[1]);
        // genericOnOffSet(onoff[1]);
}

void button2_work_handler(struct k_work *work)
{
    genericOnOffSetUnAck(onoff[0]);
        // genericOnOffSet(onoff[0]);
}

void button3_work_handler(struct k_work *work)
{
        genericOnOffGet();
}
```

Build and install this code with *west flash*.

Press button 1. You should see console output which looks similar to this:

Note the configured publish address being used as the destination address ("dst").

Note this line:

```
bt_mesh_access.model_send: len 4: 82020102
```
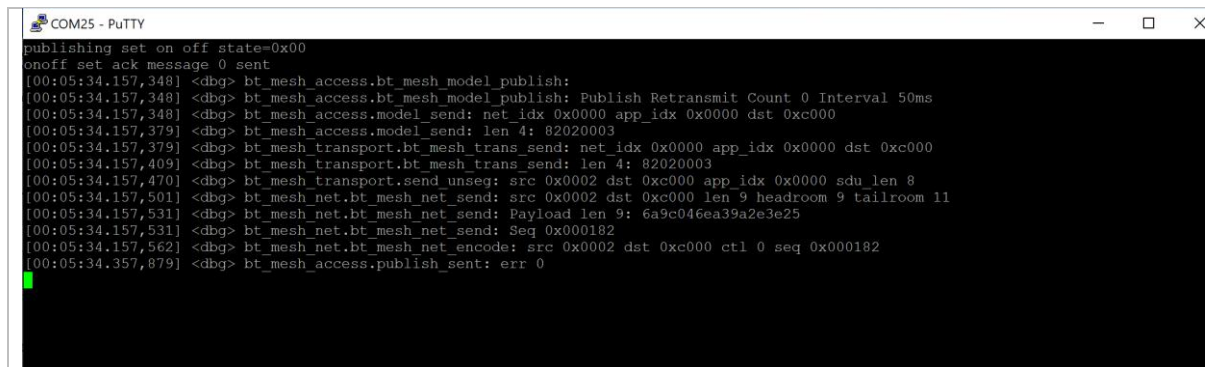
Here we can see the opcode 0x8202 of the *generic on off set* message followed by an on off state value of 0x01 and a transaction identifier of 0x02.

Press button 2 and note similar output other than the expected on off state value of 0x00.

Press button 3 to pubish a *generic on off get* message.

Note the opcode of 0x8201 for this message, which is as expected. At this stage we cannot verify that our node can receive the expected status message(s) from devices which implement the generic on off server model and which have subscribed to our group address.

on/off test 2

generic onoff set unacknowledged (ON)

generic onoff set unacknowledged (OFF)

Change the comments in your button1_work_handler and button2_work_handler functions so that instead of sending *generic onoff set* messages, they send ***generic onoff set unacknowledged*** messages.

```
void button1_work_handler(struct k_work *work)
{
    genericOnOffSetUnAck(onoff[1]);
        // genericOnOffSet(onoff[1]);
}

void button2_work_handler(struct k_work *work)
```

```
{
    genericOnOffSetUnAck(onoff[0]);
        // genericOnOffSet(onoff[0]);
}
```

Flash this new code and test by pressing buttons 1 and 2.

Note the opcode of 0x8203 which is as expected for a *generic on off set unacknowledged* message.

Next

Your next task is to move on to the next coding exercise and implement the *generic on off server model* on your light node.