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

Version	Date	Author	Changes
1.0.0	15th June 2018	Martin Woolley Bluetooth SIG	Initial version.

Exercise 3 – Implementing the On/Off Switch

Introduction

Your first coding exercise will involve implementing the code required to turn one of your devices into a Bluetooth mesh on/off switch. You'll test against another micro:bit which will act as a light. In exercise 5 you'll write the code for the light node but for now, to give you something to test against you'll install a pre-built binary hex file.

Debugging

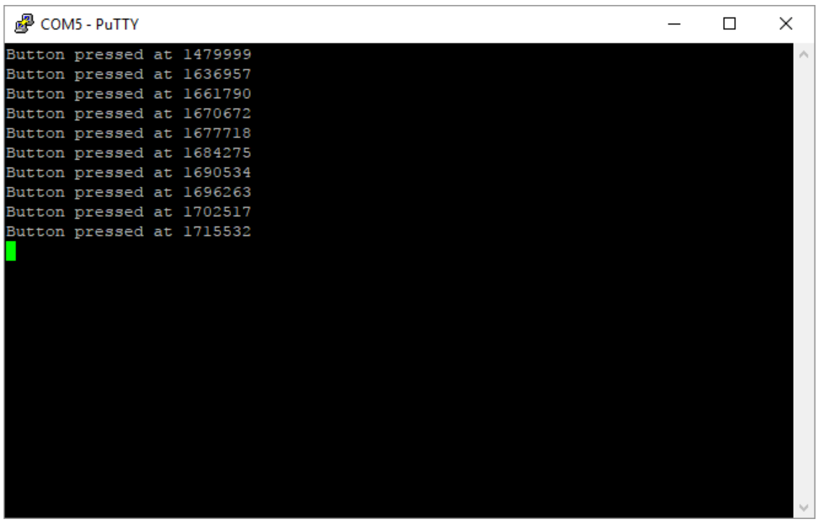
As you proceed with your coding and testing, there will be occasions where you want or need to trace execution of the code. This may be because there's a problem you need to solve or purely to reassure yourself that execution is following the path you think it is. Whatever your reasons, here are some 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 A pressed at %d\n", k_cycle_get_32());
```

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



```
Button pressed at 1479999
Button pressed at 1636957
Button pressed at 1661790
Button pressed at 1670672
Button pressed at 1677718
Button pressed at 1684275
Button pressed at 1690534
Button pressed at 1696263
Button pressed at 1702517
Button pressed at 1715532
```

tracing access messages

The Zephyr framework includes various build flags, some of which are concerned with logging. Place the following in your project's prj_bbc_microbit.conf file

```
CONFIG_BT_DEBUG_LOG=y
CONFIG_BT_MESH_DEBUG_ACCESS=y
CONFIG_BT_MESH_DEBUG=y
```

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
```

Provisioning and Configuration

Under normal circumstances, a new mesh device needs to be provisioned and configured using a smartphone application, command line tool or similar. The keys and configuration data provided to the new node in this way, would be stored persistently and restored when the device was powered up.

To keep things simple, we are not going to concern ourselves with persisting data and so to avoid the need to provision and configure each time we power our devices on, we'll use a couple of tricks called *self provisioning* and *self configuration*. This involves hard coding both the security keys which provisioning would usually provide dynamically and configuration state data which would normally come from the configuration application. This will make it easy for us to iterate through code, build, test cycles.

Important Note: *self provisioning* and *self configuration* are coding tricks and are not applicable to the creation of production or commercial devices. Before completing firmware implementation, devices must always have the ability to be dynamically provisioned and configured by a suitable client application.

Let's get started.

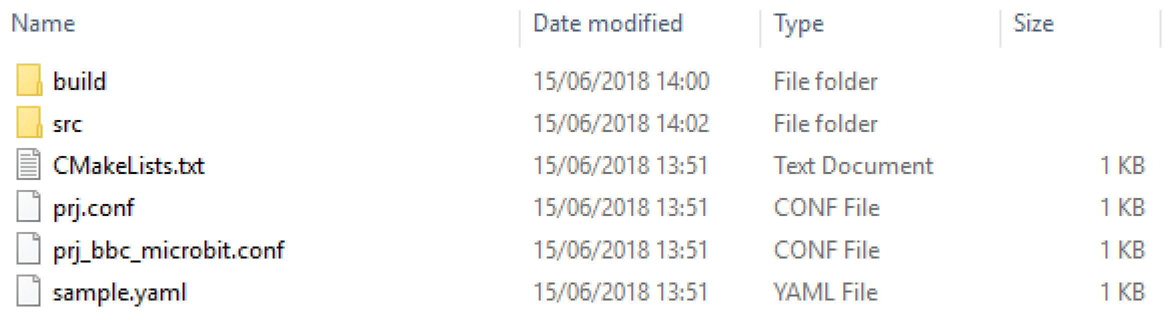
Project Set Up

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	15/06/2018 14:00	File folder	
src	15/06/2018 14:02	File folder	
CMakeLists.txt	15/06/2018 13:51	Text Document	1 KB
prj.conf	15/06/2018 13:51	CONF File	1 KB
prj_bbc_microbit.conf	15/06/2018 13:51	CONF File	1 KB
sample.yaml	15/06/2018 13:51	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.

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. Your output should be similar to this:

```
C:\workspaces\zephyr_projects\mdk_switch_solution\build>ninja
[1/138] Generating always_rebuild
Building for board bbc_microbit
[133/138] Linking C executable zephyr\zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:       59608 B       256 KB     22.74%
            SRAM:       12192 B        16 KB     74.41%
        IDT_LIST:         132 B         2 KB      6.45%
[138/138] Linking C executable zephyr\zephyr.elf
```

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 IRC channel for help.

Tracing Mesh Messages

Your project will include a file called prj_bbc_microbit.conf. Open it in your editor and note that it includes the following entries. These settings will configure the build system to include code which results in mesh messages, including access layer messages, being logged to an attached console such as a terminal connected to the serial port your micro:bit is plugged into.

```
CONFIG_BT_DEBUG_LOG=y
CONFIG_BT_MESH_DEBUG_ACCESS=y
CONFIG_BT_MESH_DEBUG=y
```

It is recommended that you have all your Zephyr projects use these settings and while developing and testing, always have your micro:bit connected to your computer over USB and 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.

Now open src/main.c and continue.

Addresses

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

```
#define NODE_ADDR 0x0001
#define GROUP_ADDR 0xc000
static u16_t target = GROUP_ADDR;
static u16_t node_addr = NODE_ADDR;
```

Security Keys

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

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

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

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

Explanation

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

Key Indices and IV Index

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

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

Miscellaneous Variables

Define the following variables under the *other* comment.

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

Explanation

tid is the transaction identifier field which transactional messages require.

flags is a Zephyr API field used when provisioning.

Configuration Client and Server Models

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

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

Explanation

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

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

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

Explanation

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

Health Server Model

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

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

Explanation

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

Generic OnOff Message Types

Under the *message types defined by this model* comment, 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 so that we can reference them more easily elsewhere in our code.

RX Messages and Handler Functions

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

Add this code 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, gen_onoff_status},
    BT_MESH_MODEL_OP_END,
};
```

Explanation

This array of *bt_mesh_model_op* types contains a single significant item, which specifies the opcode for the Generic 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 a function called gen_onoff_status which will handle all such messages received.

BT_MESH_MODEL_END indicates the end of the definition.

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

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

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

Explanation

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

Node Composition

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

Open your switch project's main.c file file and find the section with the comment heading "Composition". Add the following code under the comment:

```
static struct bt_mesh_model sig_models[] = {
    BT_MESH_MODEL_CFG_SRV(&cfg_srv),
    BT_MESH_MODEL_CFG_CLI(&cfg_cli),
    BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
    BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_ONOFF_CLI, gen_onoff_cli_op, NULL, &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 client and server models 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 definitions we prepared earlier on.

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

```
// node contains elements. Note that BT_MESH_MODEL_NONE means "none of this type" and here
means "no vendor models"
static struct bt_mesh_elem elements[] = {
                              BT_MESH_ELEM(0, sig_models, BT_MESH_MODEL_NONE),
};

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

Explanation

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

Checkpoint

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

```
C:\workspaces\zephyr_projects\mdk_switch_solution\build>ninja
[1/100] Generating always_rebuild
Building for board bbc_microbit
[2/7] Building C object CMakeFiles/app.dir/src/main.c.obj
../src/main.c:168:29: warning: 'sig_models' defined but not used [-Wunused-variable]
 static struct bt_mesh_model sig_models[] = {
                             ^~~~~~~~~~
../src/main.c:106:13: warning: 'tid' defined but not used [-Wunused-variable]
 static u8_t tid;
             ^~~
../src/main.c:104:13: warning: 'flags' defined but not used [-Wunused-variable]
 static u8_t flags;
             ^~~~~
../src/main.c:100:20: warning: 'iv_index' defined but not used [-Wunused-const-variable=]
 static const u32_t iv_index;
                    ^~~~~~~~
../src/main.c:98:20: warning: 'app_idx' defined but not used [-Wunused-const-variable=]
 static const u16_t app_idx;
                    ^~~~~~~
../src/main.c:97:20: warning: 'net_idx' defined but not used [-Wunused-const-variable=]
 static const u16_t net_idx;
                    ^~~~~~~
../src/main.c:75:19: warning: 'app_key' defined but not used [-Wunused-const-variable=]
 static const u8_t app_key[16] = {
                   ^~~~~~~
../src/main.c:55:19: warning: 'net_key' defined but not used [-Wunused-const-variable=]
 static const u8_t net_key[16] = {
                   ^~~~~~~
../src/main.c:35:19: warning: 'dev_key' defined but not used [-Wunused-const-variable=]
 static const u8_t dev_key[16] = {
                   ^~~~~~~
../src/main.c:30:14: warning: 'node_addr' defined but not used [-Wunused-variable]
 static u16_t node_addr = NODE_ADDR;
              ^~~~~~~~~
../src/main.c:29:14: warning: 'target' defined but not used [-Wunused-variable]
 static u16_t target = GROUP_ADDR;
              ^~~~~~
[4/7] Linking C executable zephyr\zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:       59608 B       256 KB     22.74%
            SRAM:       12192 B        16 KB     74.41%
        IDT_LIST:         132 B         2 KB      6.45%
[7/7] Linking C executable zephyr\zephyr.elf

C:\workspaces\zephyr_projects\mdk_switch_solution\build>
```

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.

Device UUID

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

Add the following under the *device UUID* comment:

```
// cfa0ea7e-17d9-11e8-86d1-5f1ce28adea1
static const uint8_t dev_uuid[16] = {0xcf, 0xa0, 0xea, 0x7e, 0x17, 0xd9, 0x11, 0xe8, 0x86,
0xd1, 0x5f, 0x1c, 0xe2, 0x8a, 0xde, 0xa1};
```

Bluetooth Stack Initialisation

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

Update your main function as shown:

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

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

Explanation

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

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

Bluetooth Mesh Initialisation

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

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

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

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

Checkpoint

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

```
C:\workspaces\zephyr_projects\mdk_switch_solution\build>ninja
[1/100] Generating always_rebuild
Building for board bbc_microbit
[2/7] Building C object CMakeFiles/app.dir/src/main.c.obj
../src/main.c:106:13: warning: 'tid' defined but not used [-Wunused-variable]
 static u8_t tid;
             ^~~
../src/main.c:104:13: warning: 'flags' defined but not used [-Wunused-variable]
 static u8_t flags;
             ^~~~~
../src/main.c:100:20: warning: 'iv_index' defined but not used [-Wunused-const-variable=]
 static const u32_t iv_index;
                    ^~~~~~~~
../src/main.c:98:20: warning: 'app_idx' defined but not used [-Wunused-const-variable=]
 static const u16_t app_idx;
                    ^~~~~~~
../src/main.c:97:20: warning: 'net_idx' defined but not used [-Wunused-const-variable=]
 static const u16_t net_idx;
                    ^~~~~~~
../src/main.c:75:19: warning: 'app_key' defined but not used [-Wunused-const-variable=]
 static const u8_t app_key[16] = {
                   ^~~~~~~
../src/main.c:55:19: warning: 'net_key' defined but not used [-Wunused-const-variable=]
 static const u8_t net_key[16] = {
                   ^~~~~~~
../src/main.c:35:19: warning: 'dev_key' defined but not used [-Wunused-const-variable=]
 static const u8_t dev_key[16] = {
                   ^~~~~~~
../src/main.c:30:14: warning: 'node_addr' defined but not used [-Wunused-variable]
 static u16_t node_addr = NODE_ADDR;
              ^~~~~~~~~
../src/main.c:29:14: warning: 'target' defined but not used [-Wunused-variable]
 static u16_t target = GROUP_ADDR;
              ^~~~~~
[4/7] Linking C executable zephyr\zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:      106100 B       256 KB     40.47%
            SRAM:       15544 B        16 KB     94.87%
        IDT_LIST:         132 B         2 KB      6.45%
[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.

Self-Provisioning

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

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

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

```
static int selfProvision(void)
{
  // now we provision ourselves... this is not how it would normally be done!
  int err = bt_mesh_provision(net_key, net_idx, flags, iv_index, node_addr,dev_key);
  if (err)
  {
    printk("Provisioning failed (err %d)\n", err);
    return err;
  }
  printk("Provisioning completed\n");
  return 0;
}
```

Explanation

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

Self-Configuration

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

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

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

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

  /* Add Application Key */
  bt_mesh_cfg_app_key_add(net_idx, node_addr, net_idx, app_idx, app_key, NULL);

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

  /* Bind to Health model */
  bt_mesh_cfg_mod_app_bind(net_idx, node_addr, node_addr, app_idx,
BT_MESH_MODEL_ID_HEALTH_SRV, NULL);
```

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

Explanation

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

We accomplish three things in this function.

1. Using *bt_mesh_cfg_app_key_add* we add our AppKey to the node and at the same time, bind it to the NetKey. AppKeys are always associated with or *bound to* a specific NetKey.

2. We then call *bt_mesh_cfg_mod_app_bind* to bind our AppKey to the generic onoff client model so that this key is used for securing the fields in generic onoff client messages, relating to the higher layers of the stack.

3. Finally, we then call *bt_mesh_cfg_mod_app_bind* to bind our AppKey to the health server model so that this key is used for securing the fields in health server messages, relating to the higher layers of the stack.

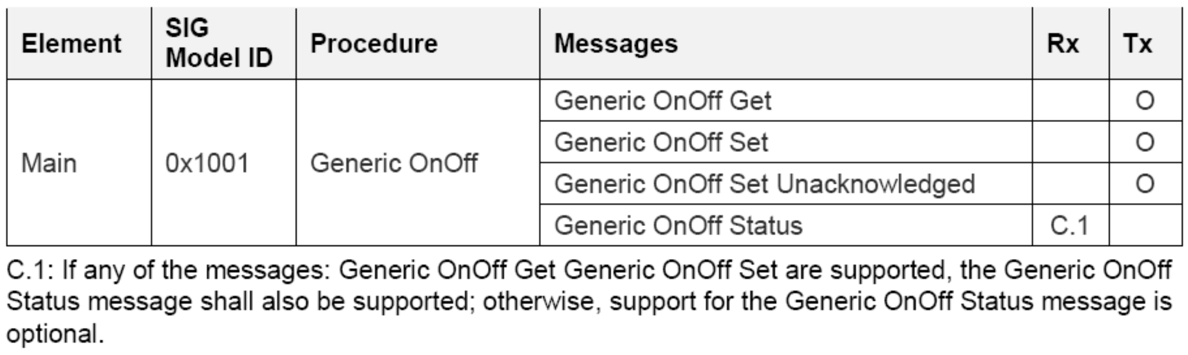
Checkpoint

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

```
C:\workspaces\zephyr_projects\mdk_switch_solution\build>ninja
[1/100] Generating always_rebuild
Building for board bbc_microbit
[2/7] Building C object CMakeFiles/app.dir/src/main.c.obj
../src/main.c:106:13: warning: 'tid' defined but not used [-Wunused-variable]
 static u8_t tid;
             ^~~
../src/main.c:29:14: warning: 'target' defined but not used [-Wunused-variable]
 static u16_t target = GROUP_ADDR;
              ^~~~~~
[4/7] Linking C executable zephyr\zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:      109948 B       256 KB     41.94%
            SRAM:       15552 B        16 KB     94.92%
        IDT_LIST:         132 B         2 KB      6.45%
[7/7] Linking C executable zephyr\zephyr.elf
```

RX Messages

Our switch implements the configuration client and server models, the health server model and the generic onoff client model. The mesh model 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, acknowledged SET messages but can also be sent at any time by the server.

Generic OnOff Status

In the *RX Messages and Handler Functions* section, we registered a function to handle generic onoff status messages and added a skeleton implementation of that function. Let's complete it now so that the micro:bit displays the value of the onoff state in the received status message. The value will always be either a 0 or a 1.

Complete the gen_onoff_status function so that it looks like this:

```
static void gen_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);
    struct mb_display *disp = mb_display_get();
    mb_display_print(disp, MB_DISPLAY_MODE_SINGLE, K_SECONDS(1), "%d", 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 gen_onoff_status function:

```
static const struct bt_mesh_model_op gen_onoff_cli_op[] = {
        {BT_MESH_MODEL_OP_GENERIC_ONOFF_STATUS, 1, gen_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_CFG_CLI(&cfg_cli),
               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 gen_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 onoff state value. We then use a micro:bit API to display the onoff state value as the character "0" or "1" on the LED display.

TX Messages

We'll now implement the three TX messages. We'll trigger sending the TX messages using the two buttons on the front of the micro:bit. For normal use we'll send generic onoff set unacknowledged messages with a value of 1 if button A is pressed and a value of 0 if button B 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 *TX message producer functions* section:

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

  NET_BUF_SIMPLE_DEFINE(msg, 2 + 0 + 4);

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

  bt_mesh_model_msg_init(&msg, BT_MESH_MODEL_OP_GENERIC_ONOFF_GET);

  if (bt_mesh_model_send(&sig_models[2], &ctx, &msg, NULL, NULL))
  {
     printk("Unable to send generic onoff get message\n");
  }
  printk("onoff get message sent\n");
}
```

Uncomment the call to *genericOnOffGet* in the *buttonB_work_handler* function and ensure calls to other functions are commented out, like this:

```
void buttonB_work_handler(struct k_work *work)
{
  printk("Button B work handler\n");
  // genericOnOffSetUnAck(onoff[0]);
  // genericOnOffSet(onoff[0]);
  genericOnOffGet();
}
```

Explanation

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

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

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

The Generic OnOff Set message types

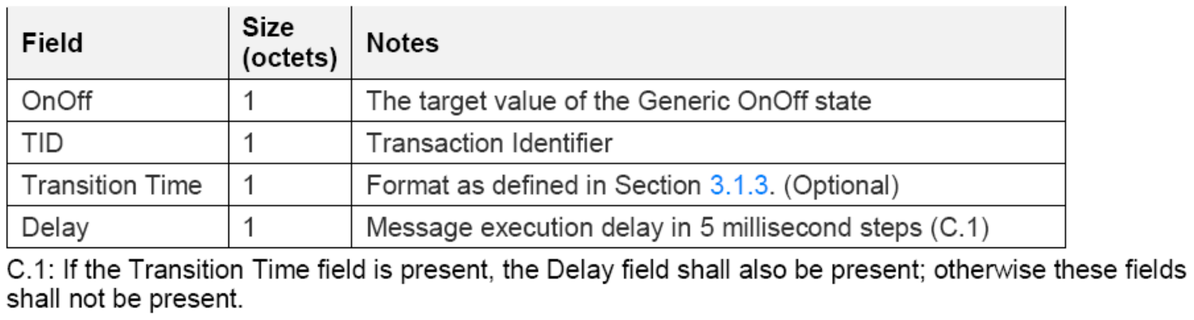
The code required to send a *generic onoff set unacknowledged* message only differs from the code used when sending a *generic onoff set* message in that the message opcode is different. Consequently we'll use a common function for both message types, with the required message opcode as a parameter. 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)
{
  // 2 bytes for the opcode
        // 2 bytes parameters: first is the onoff value, second is for the TID
        // 4 additional bytes for the TransMIC

        NET_BUF_SIMPLE_DEFINE(msg, 2 + 2 + 4);

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

        bt_mesh_model_msg_init(&msg, message_type);
        net_buf_simple_add_u8(&msg, on_or_off);
        net_buf_simple_add_u8(&msg, tid);
        tid++;

        return bt_mesh_model_send(&sig_models[2], &ctx, &msg, NULL, NULL);
}
```

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

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 unack message\n");
        }
        printk("onoff set unack 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");
        }
        printk("onoff set unack message %d sent\n",on_or_off);
}
```

Testing the On/Off Switch

We've now implemented all message types and by uncommenting required function calls in the *buttonA_work_handler* and *buttonB_work_handler* functions, we should be able to test each of them. We'll need an implementation of the generic onoff server to test against of course, and we haven't written this code yet. To that end, the binaries folder contains a hex file light/light_node.hex which implements both the generic onoff server and generic level server models. Install this hex file on one of your test devices now.

on/off test 1

generic onoff set unacknowledged (ON)

generic onoff get

generic onoff status

Adjust the statements in your buttonA_work_handler and buttonB_work_handler functions so that they look like this:

```
void buttonA_work_handler(struct k_work *work)
{
  printk("Button A work handler\n");
  genericOnOffSetUnAck(onoff[1]);
  // genericOnOffSet(onoff[1]);
}

void buttonB_work_handler(struct k_work *work)
{
  printk("Button B work handler\n");
  // genericOnOffSetUnAck(onoff[0]);
  // genericOnOffSet(onoff[0]);
  genericOnOffGet();
}
```

Build and install this code on the device which will act as your on/off switch. The binary hex file produced by the build process is in your project's build\zephyr folder and is called zephyr.hex.

```
copy zephyr\zephyr.bin e:
1 file(s) copied.
```

Pressing button A should send a ***generic onoff set unacknowledged*** message with a value of 1. Pressing button B should send a ***generic onoff get*** message and display the value returned in the resultant ***generic onoff status*** message. Put this to the test now.

on/off test 2

generic onoff set unacknowledged (ON)

generic onoff set unacknowledged (OFF)

Change the comments in your buttonB_work_handler so that instead of it sending a *generic onoff get* it sends a ***generic onoff set unacknowledged*** message with a value of 0, meaning off. Now, button A should switch your test light node on and button B should turn it off.

Press the reset button on your light node so that SEQ values start from 0 again (otherwise the light may reject messages from your switch, believing them to constitute a replay attack). Get in the habit of reseting devices for each test cycle.

Build, install and test now.

```
void buttonA_work_handler(struct k_work *work)
{
  printk("Button A work handler\n");
  genericOnOffSetUnAck(onoff[1]);
  // genericOnOffSet(onoff[1]);
}

void buttonB_work_handler(struct k_work *work)
{
  printk("Button B work handler\n");
  genericOnOffSetUnAck(onoff[0]);
  // genericOnOffSet(onoff[0]);
  // genericOnOffGet();
}
```

on/off test 3

generic onoff set (ON)

generic onoff set (OFF)

generic onoff status

Change the comments in your buttonA_work_handler so that it sends a ***generic onoff set*** message with a value of 1, meaning on and buttonB_work_handler so that it sends a ***generic onoff set*** message with a value of 0, meaning off. Now, button A should switch your test light node on and button B should turn it off and in each case, your switch device should receive a **generic onoff status** message containing the resultant state value. Build, install and test now.

```
void buttonA_work_handler(struct k_work *work)
{
  printk("Button A work handler\n");
  // genericOnOffSetUnAck(onoff[1]);
  genericOnOffSet(onoff[1]);
}

void buttonB_work_handler(struct k_work *work)
{
  printk("Button B work handler\n");
  // genericOnOffSetUnAck(onoff[0]);
  genericOnOffSet(onoff[0]);
  // genericOnOffGet();
}
```

Next

You should now have a working on off switch. Your next task is to move on to the next coding exercise and implement a dimmer node.