TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Paulo Alexandre dos Santos Zacarias

DEVELOPMENT OF BLUETOOTH MESH APPLICATIONS FOR SMART BUILDINGS

Master thesis

Supervisor: Andres Rähni
MSc

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Paulo Alexandre dos Santos Zacarias

BLUETOOTH MESH TEHNOLOOGIA RAKENDAMINE ARUKATES HOONETES

Magistritöö

Juhendaja: Andres Rähni
MSc

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Paulo Alexandre dos Santos Zacarias

31.12.2018

Abstract

Wireless mesh networks started to gain popularity with the growth of Internet of Things (IoT) and nowadays there are already some successful technologies implementing such topology using low-cost, low-power, wireless devices.

While Bluetooth technology has always been a popular solution for audio streaming and wireless connectivity between wearables devices, it is now, with the support for mesh networking, that it promises to play a big role in IoT. This new mesh capability enables many-to-many device communications in large-scale networks which makes it more suitable for building automation, sensor networks, smart lightning and other IoT solutions.

The aims of this thesis were to explore some of the main Bluetooth mesh concepts and to introduce the development of Bluetooth mesh applications for smart buildings. The thesis contains a brief introduction to some of the most important Bluetooth mesh concepts, presented along with some practical examples.

A light control application is implemented. Its development covers the use of hardware drivers, a modified and a newly created mesh model, the handling of mesh messages and the configuration of multiple models and elements in a node. The thesis also illustrates a potential implementation of a sensor-driven control system in a Bluetooth mesh network, that uses a combination of different models to interact with sensors and use them to control other devices.

This thesis is written in English and is 55 pages long, including 6 chapters, 25 figures and 4 tables.

List of abbreviations and terms

ANT	Advanced and Adaptive Network Technology
API	Application programming interface
BLE	Bluetooth Low Energy
BR	Bluetooth Basic Rate
DK	Development Kit
EDR	Enhanced Data Rate
GATT	Generic Attribute
GFSK	Gaussian Frequency Shift Keying
GHz	Gigahertz
GPIO	General Purpose Input / Output
HVAC	Heating, ventilation, and air conditioning
I2C	Inter-Integrated Circuit
IDE	Integrated development environment
IoT	Internet of Things
IPv6	Internet Protocol version 6
ISM	Industrial, scientific and medical radio band
LED	Light-emitting Diode
PDU	Protocol Data Unit
PSK	Phase-shift keying
PWM	Pulse-width Modulation
QoS	Quality of Service
RGB	Red Green Blue
RTT	Real Time Terminal
SAADC	Successive approximation analog-to-digital converter
SDK	Software Development Kit
SES	SEGGER Embedded Studio
SPI	Serial Peripheral Interface
TTL	Time to Live

Table of contents

List of figures

List of tables

Introduction

Mesh network is a network topology with decades of existence, but it was not immediately used in a large scale, mostly because in the past, each node had to be connected through wire to other nodes, making this topology expensive and complex to set up.

However, with the advance of wireless technology, radio devices become a low-cost alternative to wired systems, making wireless mesh networks a feasible solution and very attractive for Internet of Things, especially considering the rapidly growing number of connected devices.

Smart buildings can highly benefit from the wireless mesh networks. Although many buildings already have some sort of “smart” system, like fire detection, climate control, lightning, video surveillance, etc, in most cases those systems work independently. With the use of wireless mesh networks all systems can integrate into one, and devices can communicate with other devices that are not in radio range.

Also, for buildings where tenants and their requirements might change frequently, wireless mesh networks offer the flexibility and low-cost that wired systems cannot provide.

1.1 Motivation

Bluetooth is a technology with more than 20 years of existence. Initially designed as a wireless alternative to RS-232 data cables, it is now present in many of our everyday electronic devices, like smartphones, headphones, speakers, smartwatches, computers and many others [1].

The protocol itself operates in the unlicensed ISM band centred at 2.4 GHz and can use up to 79 Bluetooth channels. It supports a data rate from 0.7 to 3 Mbit/s and has a range from 10 to 400 meters, using GFSK and PSK modulations.

Since its first release, Bluetooth has continuously evolved by the addition of new features and improvements at each revision. The first notable update was the release of the version 2.0 +EDR, which increased the maximum data transfer rate to 3 Mbit/s. This release was followed by version 3.0 that came with the possibility of using an 802.11 Wi-Fi radio link for high-speed data transfer, while still using Bluetooth protocol for discovery, connection, and configuration [1].

But it was with the release 4 that Bluetooth made its first steps into IoT. While the version 4.0 introduced the low-energy protocol, which makes it suitable to use with sensors that often run on small batteries, the release 4.2 introduced support for IPv6 for direct internet access. After that, Bluetooth version 5 was also released, coming with longer range and higher data rate than the previous version as well as increased data broadcasting capacity.

Now, Bluetooth promises to revolutionize the IoT with the addition of mesh networking support. When it comes to topologies, Bluetooth BR/EDR only supports point-to-point connection between two devices and Bluetooth Low Energy supports point-to-point and one-to-many device communication (broadcasting), but the latest specification finally brings the mesh networking capability to Bluetooth technology allowing many-to-many device communication in large scale networks [2, p. 10]. Figure 1 shows an illustration of the topologies currently supported by Bluetooth technology.

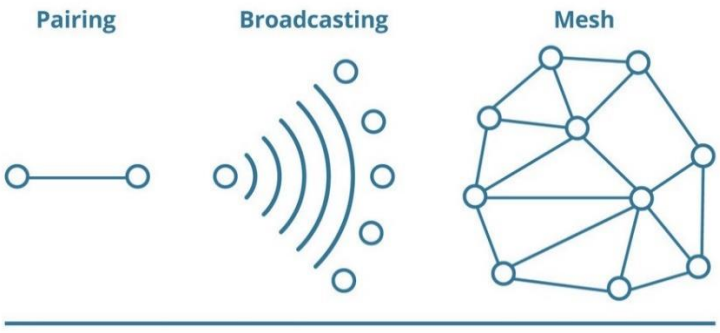

Figure 1: Topologies supported by Bluetooth technology [3].

Bluetooth mesh is especially suited for building automation, sensor networks, and other IoT solutions that require hundreds or thousands of devices to communicate in a reliable and secure way [4].

Its specification was designed with commercial lighting systems in mind, covering from radio communications up to the specific application behaviour that a product should exhibit, which guarantees interoperability across manufacturers [2, pp. 13-14].

Lighting systems can now be truly smart, allowing smartphone applications to guide people inside a building, help locating physical assets, collect and make use of data from sensors, etc. A new range of possibilities is open.

1.2 Purpose and overview of the thesis

The purpose of this thesis is to present some of the major properties of Bluetooth mesh technology and to introduce the development of Bluetooth mesh applications for smart buildings by providing two sample applications.

Besides this introduction, this thesis consists of four chapters and conclusion. Chapter 2 contains a brief description about the hardware and software development kit chosen, chapter 3 covers some of the most important aspects of the Bluetooth mesh specification, chapter 4 describes the implementation of a light control application and chapter 5 a generic solution for a sensor-driven control system.

2 Development platform

To choose the development platform to use in this thesis work, some requirements were considered. The first and fundamental requirement, was that the development kit should support, at least to some extent, the new Bluetooth mesh protocol stack. Secondly, the IDE and SDK should be preferably non-paid or free for educational use. And finally, the price, considering that, in order to explore the possibilities of a mesh network, it seems reasonable to use at least three devices.

Some options were evaluated and are briefly described in the table 1 below.

Table 1: Analysis of the different development platforms.

Manufacturer	**DK (SDK)**	**Advantages**	**Disadvantages**	**Price**
Nordic semiconductors	nRF52 DK (nRF5 SDK, nRF5 SDK Mesh)	- Some support to development in Android; - Forum for support + good documentation; -Support different IDE.	- SDK do not support 100% Mesh, yet (Friend and Low Power features missing).	≈30€
STMicroelectronics	STEVAL-IDB008V2 (STSW-BNRG-Mesh)	- Android SDK; - Some sensors on the DK.	- Less user LEDs and buttons.	≈60€
Silicon Labs	SLWSTK6020B (Mesh SDK with Simplicity Studio IDE)	- Friend and Low Power features; - More model features.	- High price.	≈130 €
Cypress	BCM92073'xxx' (different kits) (WICED SMART SDK)	- Full Mesh support (not confirmed).	- Several different DK and software tools for different purposes (hard to select); -Few information without registration.	≈36€ to ≈75€ (*)

Silicon Labs option was excluded due to the high price of its DK. Cypress did not provide much detailed information regarding the support for Bluetooth mesh and its IDE and SDK. Plus, it was not clear which, from their different DK, was compatible with Bluetooth mesh.

The choice was finally made between Nordic semiconductors and STMicroelectronics. Both options seemed equally good options, and the decision factors in the end were the available documentation and the lower price of the Nordic DK.

2.1 Development Kit

The nRF52 Development Kit board (PCA10040) is based on the nRF52832 SoC, which incorporates a 32-bit ARM Cortex M4F microcontroller unit and a multi-protocol 2.4GHz radio transceiver. It supports Bluetooth low energy, ANT and 2.4GHz proprietary protocol stacks.

Some key features and specifications [5] of the nRF52832 SoC are:

- 512kB flash and 64kB RAM memory;
- Multi-protocol 2.4GHz radio;
- Programmable output power from +4dBm to -20dBm;
- 8/10/12-bit resolution SAADC (14-bit resolution with oversampling);
- Digital interfaces: SPI/2-wire/UART/PDM/I2S;
- Supply voltage range: 1.7 V to 3.6 V;
- 8 analog inputs;
- Low power 32MHz crystal and ultra-low power 32kHz crystals and RC oscillators;
- 32 configurable GPIO pin;
- Frequency band 2.4GHz (2360 – 2483.5MHz);

Figure 2 contains an illustration of the Development Kit board.

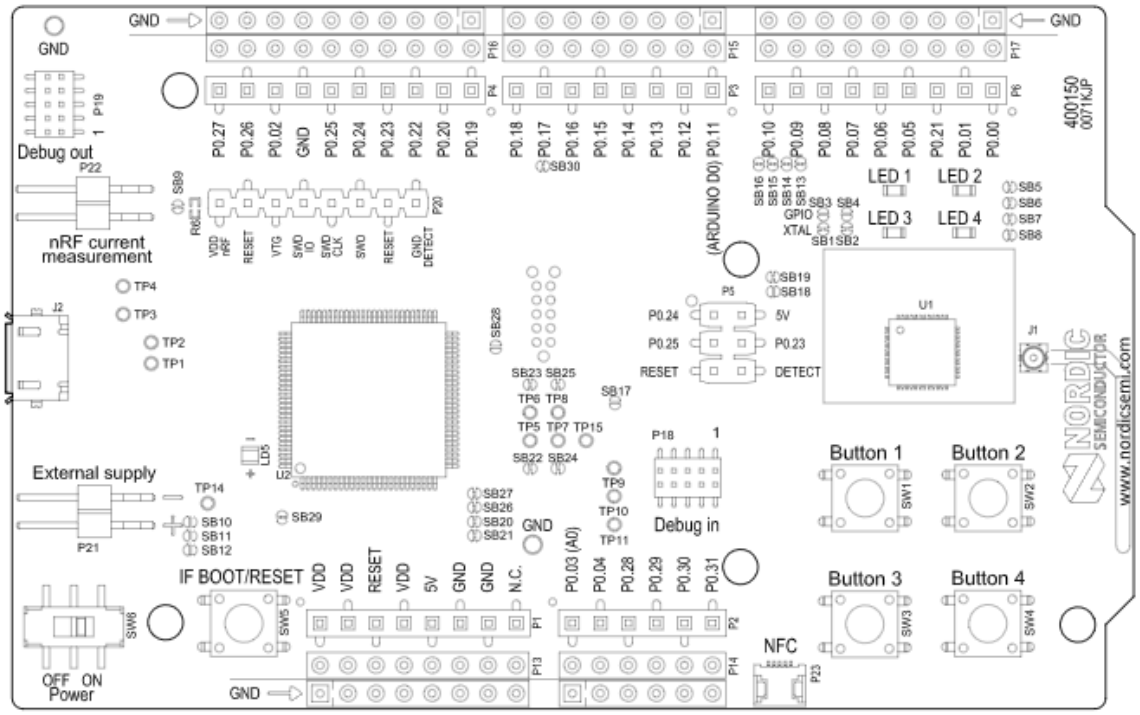

Figure 2: Illustration of the nRF52 Development Kit [6].

The board is compatible with Arduino Uno Rev3 standard allowing 3rd party Arduino Uno compatible shields to be connected to it.

It has 4 x LEDs and 4 x buttons that are user-programmable, a PCB antenna and a coin-cell battery holder, plus external connectors for NFC antenna and for RF and power consumption measurements [7].

2.2 Software Development Kit

The software support for the nRF52832 SoC consists of two parts: SoftDevice and Software Development Kit (SDK).

The concept of *SoftDevice* is intended to separate between application code and Nordic's embedded protocol stacks, by providing precompiled and linked binary software to implement the wireless protocol, Bluetooth low energy or ANT. Development of Bluetooth mesh applications for the nRF52832 is supported by the S132 SoftDevice [8].

Nordic has two software development environments for the nRF52 Series, the nRF5 SDK and the nRF5 SDK for Mesh. Both are available in .zip-file format, offering the possibility to use any of the supported IDE and compiler.

The nRF5 SDK provides a selection of drivers, libraries and examples for peripherals, for nRF5 Series chips [9], while the nRF5 SDK for Mesh provides an implementation of the Bluetooth Mesh stack as well as some example applications. Since the release 2.0.0, the nRF5 SDK for Mesh requires the nRF5 SDK 15.0.0 to compile.

2.2.1 Brief introduction to the Simple OnOff Model and the light switch demo – nRF5 SDK for Mesh

The Simple OnOff is a vendor-specific model provided in the nRF5 SDK Mesh as an introductory example for creating custom mesh models. It allows to control a single on/off state on a server by implementing the messages and corresponding behaviour required to set a 1-bit value state, that can be used to switch things like a light lamp, for example.

This model can be considered as a simplified version of the Generic OnOff Model specified in the Mesh Model Specification v1.0 which provides additional features such as control over when and for how long the transition between the on/off state should be performed [10]. Its functionality is achieved using two parts: the server model, maintaining the OnOff state, and a client model, used for manipulating the OnOff state on the server.

The supported messages in this model are:
Simple OnOff Acknowledged Set – Reliable message sent from the Client to Set the on/off state in the Server. In response to this message Server should reply with a Simple OnOff Status message.
Simple OnOff Get – Message used to get the current on/off state of the Server.
Simple OnOff Set Unreliable – Unreliable message sent from the Client to Set the on/off state in the Server. No reply message is needed from Server side.
Simple OnOff Status – Message used by the Server to inform Client about a change in its state due to a local event or in response to an Acknowledged Set message.

The light switch demo project is an implementation of the Simple OnOff model designed to control a single LED on the development board. It consists of three examples: the light switch server, the light switch client, both with and without GATT proxy support, and a provisioner example [11].

2.2.2 Developing with SEGGER Embedded Studio

Nordic Software Development tools support different IDEs and compilers, like Keil, IAR, GCC and SEGGER Embedded Studio (SES).
For this thesis work the IDE used was the SES, which is free (without any limitations) [12] for any non-commercial use and offers full debug support, including Real Time Terminal (RTT).

After installing the most recent releases of SES (Embedded Studio for ARM) and the J-Link Software and Documentation Pack, from the SEGGER download page [13], one can start developing with the nRF5 SDK for Mesh by opening a SES project file from the examples folder, like:

\...\nrf5_SDK_for_Mesh_v2.1.1_src\examples\light_switch\server\light_switch_server_nrf52832_xxAA_s132_6_0_0.emProject

To program the first application on the development board the contents of the board should erased. This is done following the steps shown in figure 3:

1. Selecting **Target → Connect J-Link.**
2. And after the connection **Erase All**

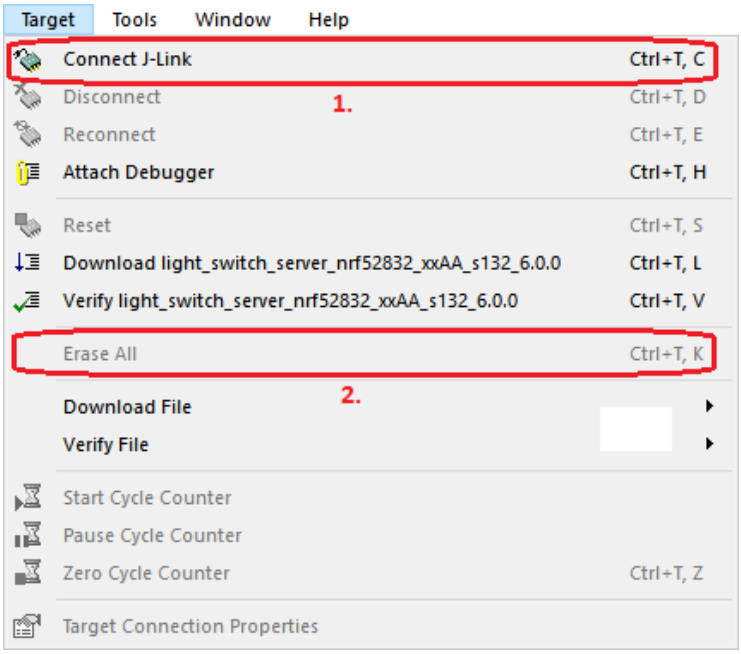

Figure 3: Steps to erase the flash memory of the Development Kit board.

When working with Bluetooth mesh is important to note that the device keeps information about the network in the flash memory. So, this step should not be done if the device is supposed to keep the same role in the network after re-programming, i.e. if the models and elements do not change and only the application changes.

Compiling the application from a SES project is done by selecting **Build→ Build "project_target_name"**, or alternatively by pressing **F7**.
If there are no build errors, the program can be loaded to the board by selecting **Debug→ Go** or alternatively by pressing **F5**. This will load the program, to start running it from main loop, the previous step must be repeated (**Debug→ Go** or **F5**).

Figure 4 shows some useful tools to use when debugging an application with SEGGER Embedded Studio:

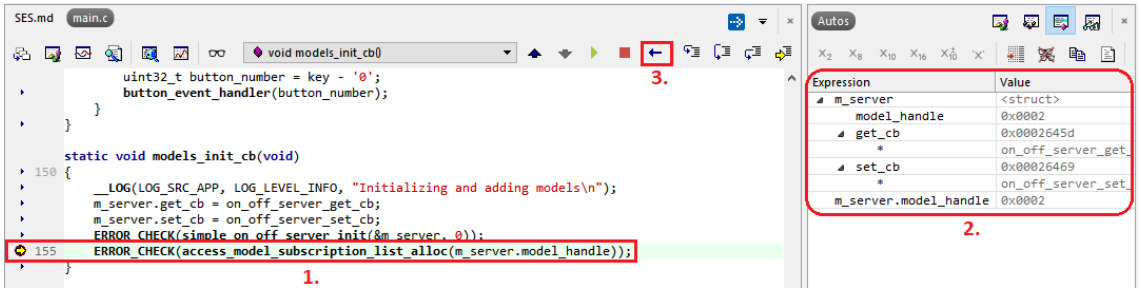

Figure 4: Debugging an application with SEGGER Embedded Studio.

It is possible to set break points (1) through the application, read registers (2), and so on. It might be necessary to restart the application when in debugging mode, in that case it's not necessary to restart the debugger but rather the application itself by pressing the restart button (3). That will restart the application with a breakpoint in the main loop, to continue debugging the **F5** key must be pressed.

3 Bluetooth mesh

The Bluetooth mesh is a specification released in July of 2017 which allows many-to-many communication over Bluetooth, meaning that devices can communicate with multiple other devices within the network even if they are not in direct radio range, thus increasing the range of Bluetooth networks [14].

The Bluetooth Mesh is based on the Bluetooth Low Energy and shares the lowest layers with this protocol (Link and Physical Layers) [15]. However, Bluetooth Mesh specifies a completely new layered system architecture [16] [17], shown in Table 2.

Table 2: Layered architecture of Bluetooth mesh.

model layer	→	Defines models and thus the functionality and behaviour of a node.
foundation model layer	→	Defines states, messages and models required to configure and manage a mesh network.
access layer	→	Defines format of application data and how higher layer applications use the upper transport layer.
upper transport layer	→	Encrypts, decrypts and authenticates application data to provide confidentiality of access messages.
lower transport layer	→	Defines how messages are segmented and reassembled into multiple lower transport PDUs, when required.
network layer	→	Defines how messages are addressed to one or more elements. Implements the relay and proxy features.
bearer layer	→	Defines how network messages are transported between nodes using the underlying BLE stack.
Bluetooth Low Energy	→	Bluetooth Low Energy Core Specification (Link layer and Physical layer).

In a general way, the mesh network operation is designed to enable messages to be sent between elements, allow nodes to relay received messages to extend the range of communication and secure messages against known security attacks. It was is also meant to work on existing devices in the market today, by making use of the BLE lower layers [18, p. 17].

Bluetooth mesh also implements the concept of state values in a client-server architecture. A state is a certain type of value that represents a condition of an element and is associated with some behaviour.For example, a light which may either be on or off.

Bluetooth mesh defines a state called Generic OnOff, in which the state value of On would cause the light to be illuminated and the state value of Off would cause the light to be switched off [19, p. 11]. The function of a server is to expose the state of the light, while the client access or changes the state of the server.

Messages are the mechanism by which a state or of multiple states can be set. There are three basic types of messages, GET, SET and STATUS, which define the three main types of operation that Bluetooth mesh supports.

GET messages are used to request the value of a given state, SET messages are used to change the value of a state and can be acknowledged or unacknowledged, and STATUS messages are a response to a GET or acknowledged SET [19, p. 12]. Status messages can also be spontaneous messages sent as result of a local event, like a periodic sensor reading.

Every message is identified by an operation code (opcode) which may comprise 1 or 2 octets, for Bluetooth SIG defined applications or 3 octets for manufacturer-specific opcodes [18, p. 93].

3.1 Bluetooth mesh concepts

In this section are described some of the main concepts of the Bluetooth mesh. It is not an extensive analysis of all the technical aspects of the mesh specification, but rather an overview of those that might help the development of Bluetooth mesh applications.

Along with the theory, are included some practical examples that might help to explain those concepts.

3.1.1 Addresses

Bluetooth mesh defines three types of address: unicast address, virtual address and group address. There is also a specific value to represent an unassigned address [18, p. 22].

A unicast address represents a single element of a node and is assigned during the provisioning process. There are 32767 unicast addresses available per mesh network. A group address is a multicast address which represents multiple elements on one or more nodes. There are 16384 group addresses available per mesh network. A virtual address is also a multicast address that can represent multiple elements but takes the form of a 128-bit UUID label [19, p. 10].

The address ranges are distributed as follow:

- 0×0000 is undefined/unassigned;
- 0×1000 – 0×7FFF Unicast addresses;
- 0×8000 – 0×BFFF Virtual addresses;
- 0×C000 – 0×FFFF Group addresses;

3.1.2 Provisioning

Provisioning is the process of adding a device to a mesh network. During the provisioning process the device receives provisioning data that allows it to become a mesh node. The process is managed by a device called provisioner. [20].

Provisioning data includes a network key, a unicast address for each element and unique ID for the device being added. The device must store this provision information permanently so that it can re-join the network after a power cycle.

Provisioner is a device that establishes a secure communication with an unprovisioned device to exchange the provisioning data. Typically, it will be a smart phone or other mobile computing device [18, p. 227].

Regarding the Nordic Mesh SDK, there are two possible ways to provision a device, using a DK board with the provisioning application [11] provided in the light switch example or using the nRF Mesh mobile application.

The provisioning example is a fast way to provisioning multiple devices. After starting the provisioning process, all unprovisioned devices will be automatically provisioned one by one. This is particularly good for networks with a larger number of devices, but it requires significant changes in the code to use it with different applications than the one provided in the example.

The mobile application is and easier and more flexible way of provisioning a device. The disadvantage is that it requires the devices to be manually provisioned one by one, which becomes unpractical for a larger number of devices. The figure 5 presents the options to add (provision) a node to the mesh network.

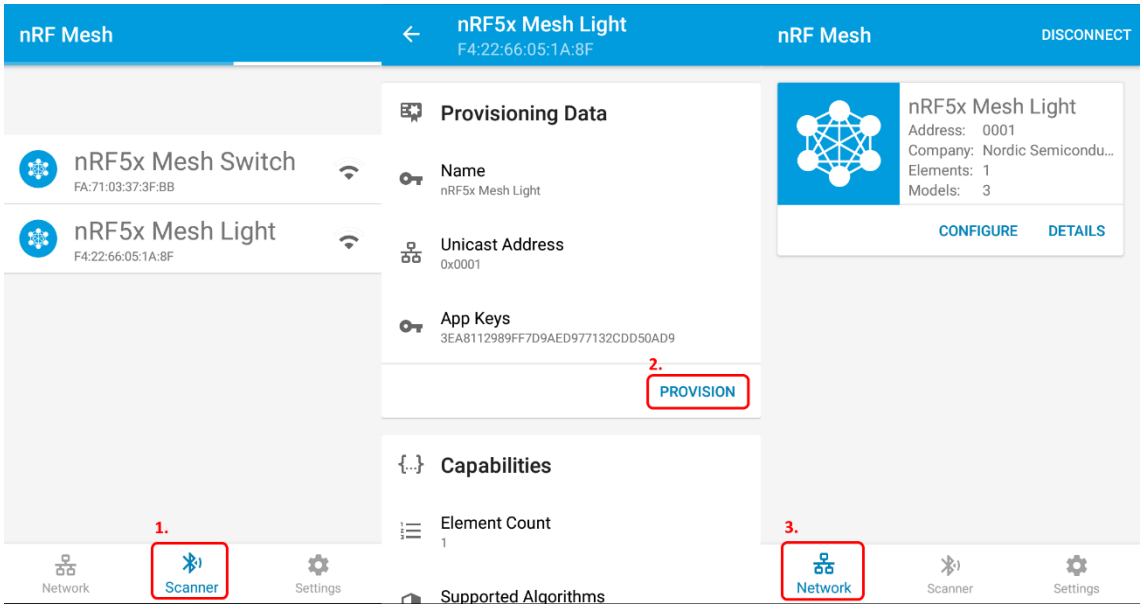

Figure 5: Options to provision mesh nodes using the nRF Mesh mobile application.

Provisioning a device using the nRF Mesh application, can be done by selecting “Scanner” (1), that will show all the unprovisioned devices. Then, selecting the device to add to the network and choosing “identify”, will show the provision data and the device capabilities. Choosing “provision” will start the provision process (2). If the process finishes successfully, it will show the message “Mesh Node has been successfully configured” and the device should then appear in the network (3).

At this point the device is already a node in the mesh network, however, it needs further configuration to communicate with the other nodes within the network, like adding the App Key and publish/subscribe address to the elements on the node, if applicable.

3.1.3 Publish/Subscribe

Bluetooth mesh uses a client-server architecture with a message-oriented communication based on the publish/subscribe paradigm.

Sending messages is referred as Publishing, while configuring a node to receive certain messages is known as Subscribing. A node may publish unsolicited messages to inform about changes in its status, using for that the publish address as the destination address. It may also publish messages in reply to received messages, and in this case, it uses the message originator's source address as the destination address [21].

Figure 6 shows a network for an illumination system, in which the nodes are switches and light bulbs. The switch #3, publishes messages to the group "Hallway", since the light bulbs #4 and #5 subscribe to that group, they will react to those messages, while the other bulbs will ignore them.

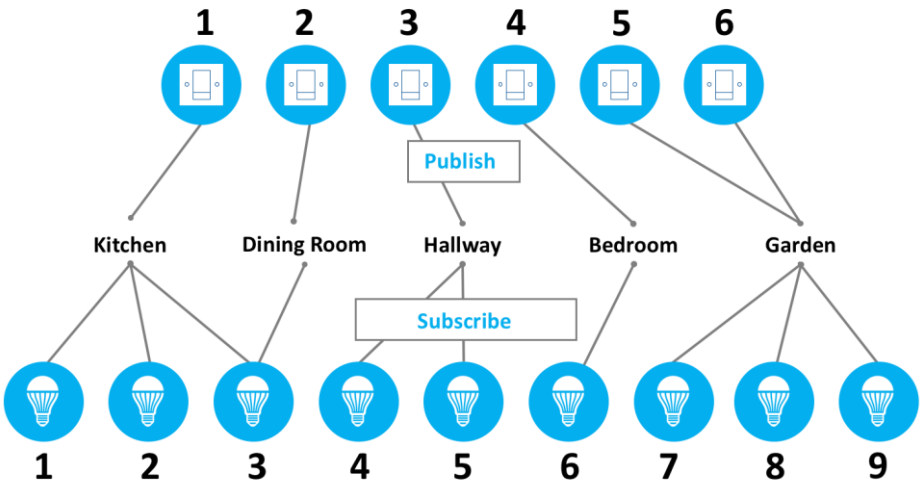

Figure 6: Illumination system using Publish/Subscribe [14].

A node may also subscribe to receive messages from one or more groups or virtual addresses. In the example of figure 6, the light #3 subscribes to both "Kitchen" and "Dining room" group address, meaning that it will receive and react to messages from both switch #1 and #2.

Some practical examples can be done using the light switch example from Nordic and the nRF Mesh provisioning app. First and most simple example is how to configure a switch to control a light. For this the client board must be configured to publish to the server's address, as shown in picture 7.

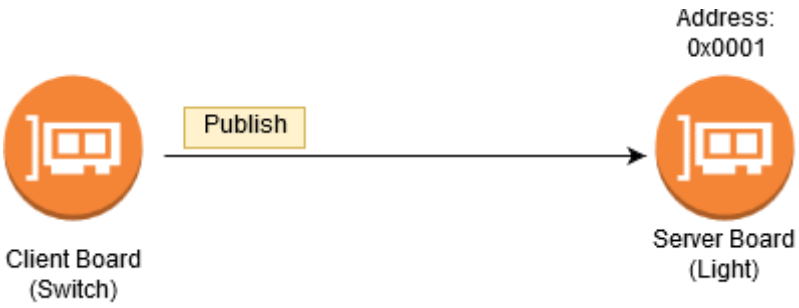

Figure 7: Example of a client publishing to the unicast address of a server.

If the nodes are already provisioned, the configuration can be done by choosing "configure" in the network view (1), as shown in figure 8. Then selecting the second element with the address 0x0004 (2) and binding an App Key to it (3). Finally, the publication address (4) has to be set to the server's address (0x0001).

On the server side, the App Key must be added to the vendor model on the first element.

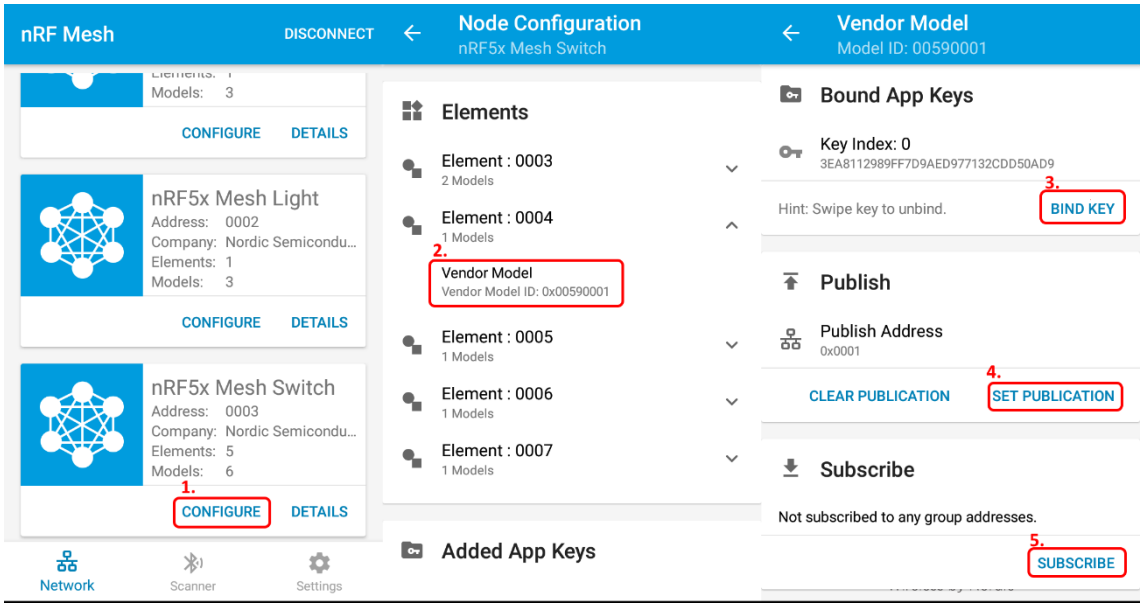

Figure 8: How to configure publish/subscribe address on a node using the nRF Mesh mobile application.

Now, pressing button 1 in the client board will cause the LED 1 in the server's board to turn On or Off. The LED 1 on the client board will change its status according to the server status, this is because the server publishes a message in reply to the received message from client. If the server does not respond to client message, the status of the LED 1 on client's board will remain unchanged.

Another example is how to control two or more lights with one switch. This is done by configuring the client to publish to a group address instead of a unicast address, as shown in figure 9.

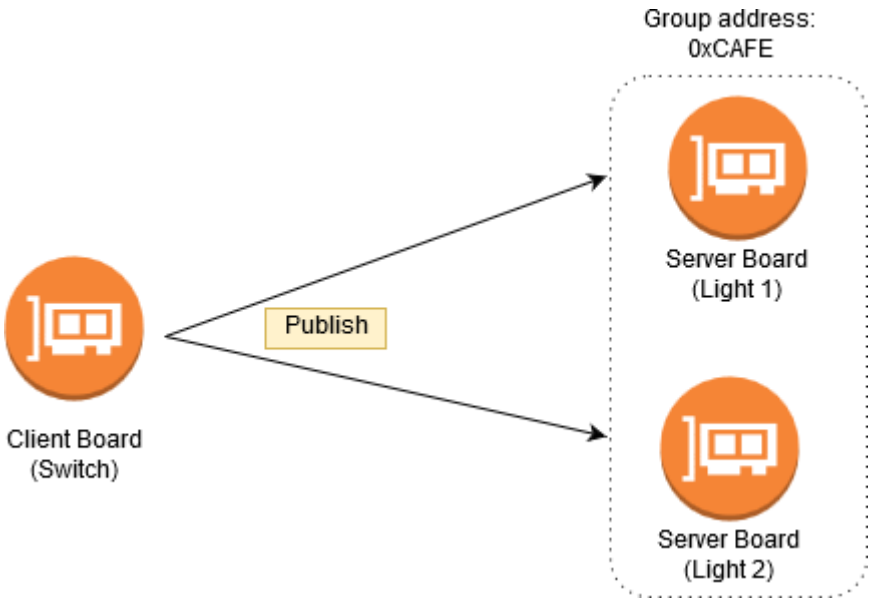

Figure 9: Example of a client publishing to a group address.

To configure the client, the steps are the same as before, but this time the publication address must be set to the group address 0xCAFE, on the client element 0x0006. Then, the servers have to subscribe to the group address 0xCAFE, (option n. 5 on figure 8).

Now, pressing button 3 on the client board will cause the LED 1 in both servers to turn On or Off simultaneously.

Third example, shown in figure 10, is having two switches controlling same light. If one switch changes the state of the light, the light should inform about its new status to the other client as well.

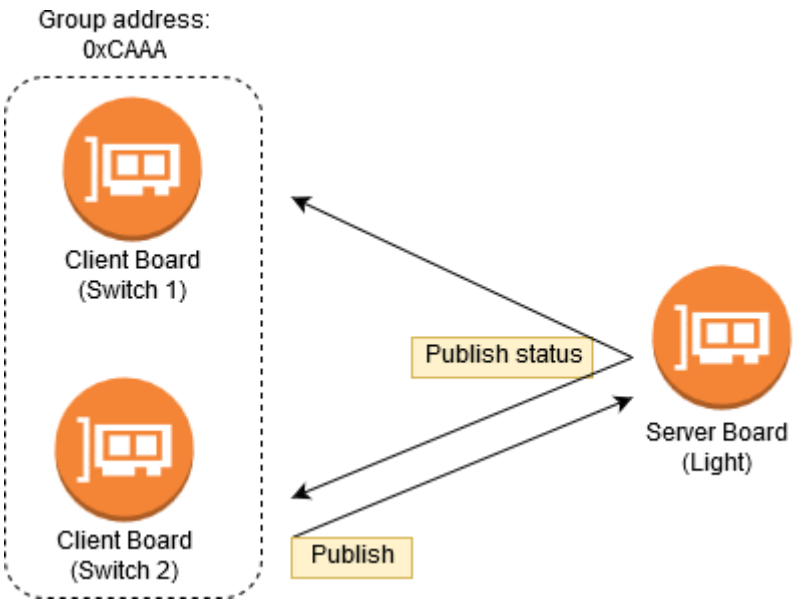

Figure 10: Example of a server publishing its status for more than one client.

The solution is to configure the server to publish its status to a group address and to configure both clients to subscribe to that same group address.
This can be done by repeating the steps done for client 1 on the client 2, so that the second client publishes to the address 0x0001 and then adding the subscription to the group address 0xCAAA on both clients and then finally by defining the publication address to the group 0xCAAA, on server side.

Now, when pressing button 1 in one of the client boards, the LED 1 on the server will turn on or off and a group message will be sent to both switches, informing about the status change. This can be confirmed by looking at the LED 1 on both clients, its status should reflect the status of the LED on the server board.
The same way, turning on or off the light locally, by pressing button 1 on the server's board, will cause the light to publish a message to the group address and both switches should update the status of their LED 1 accordingly.

3.1.4 Models and Elements

An element is an addressable entity within a node. A node must have at least one element, the primary element, and may have one or more additional secondary elements, as shown in figure 11 [22].

Each element within a node has a unique unicast address that is used to identify which element within a node is transmitting or receiving a message [18, p. 21].

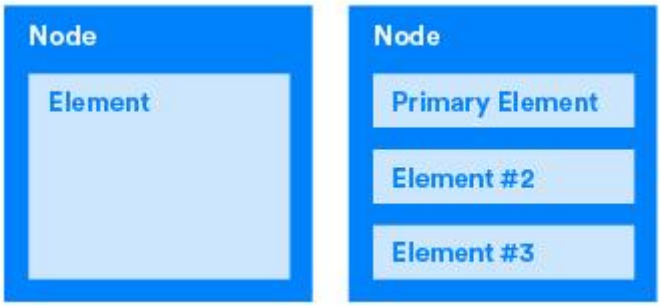

Figure 11: Example of nodes containing one or more elements [22].

The basic functionality of a node is defined using models. A model defines the behaviour of a node depending of the purpose of the device, for example, sensor readings or light control, and defines the set of messages to act on them. The Mesh Profile Specification and the Mesh Model Specification define a set of models, but vendors can define their own models and respective messages and states [15].

For example, the simplest model is the Generic OnOff Server model, which consists of two parts, the Generic OnOff Server and the Generic OnOff Client. The server reports if it is either on or off, while the client works as a binary switch and is able to control the server by sending it messages [18, p. 20].

A node may include multiple models and every element must include one or more models, as demonstrated in figure 12. Models are identified by a model ID and messages are resolved within models based on opcodes and element addresses.

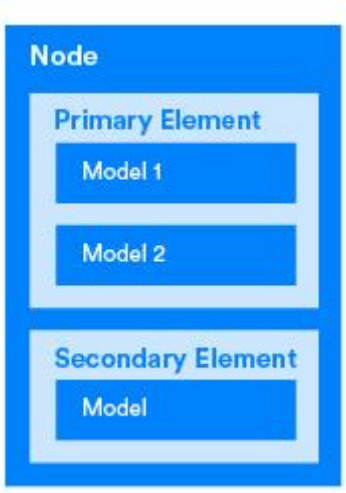

Figure 12: Example of elements containing one or more model instances [22].

A practical example of a node using more than one element, can be a light fixture with two lights or two sets of lights that must be turn On or Off independently.
The solution for this is to have two elements within the same node implementing an instance of an On-Off model each. This way only one Bluetooth radio is needed to control both lights or set of lights.

On the light switch example, the server has a single element containing an instance of the Simple On-Off model[1] (shown as Vendor Model in picture 13). This element is associated with the first LED on the board.

[1] Besides the Configuration and Health servers [18, pp. 194, 221].

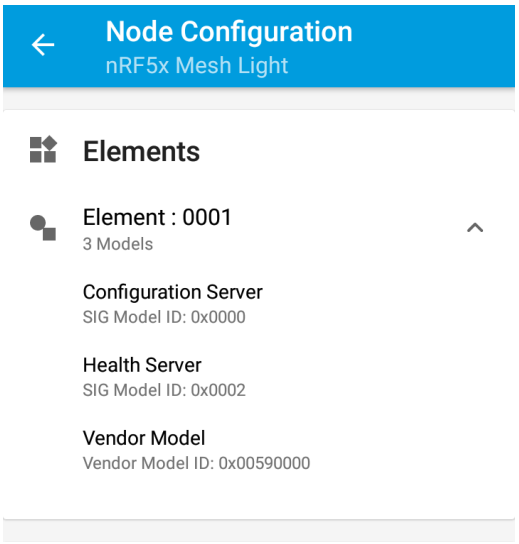

Figure 13: Structure of a server node from the light switch example application.

With some simple changes on the code, it's possible to have a second (or more) elements to control the other LEDs as well. Figure 14 shows the server now with two elements, each containing an instance of the Simple On-Off model. In appendix 1, are shown the changes and code added to the light switch example to increase the number of elements on the server.

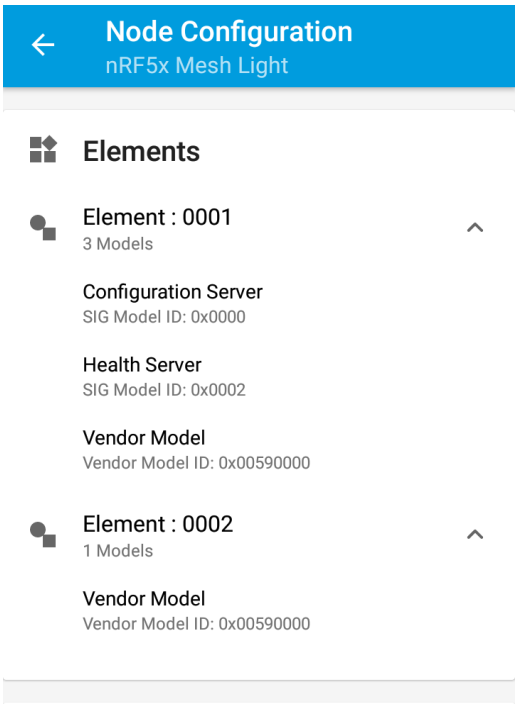

Figure 14: Structure of the server with two elements containing an instance of the Simple On-Off model each.

After implementing the changes on the server, the node configuration has now two elements, so the client can publish to the unicast address 0x0001 and 0x0002, using button 1 and 2, to respectively control LED 1 and LED 2 on server board.

3.1.5 Relay and managed flooding

To increase the range of the network, the Bluetooth mesh includes a feature that allow a node to act as a "relay". Devices with the Relay feature enabled, retransmit messages that they receive from other devices, allowing communication between devices that are not in radio range between them [17]. In the mesh network there are no dedicated relay devices and any mesh device can be configured to act as a relay [15].

Bluetooth mesh networking does not use any kind of routing mechanisms, instead, it uses a concept known as "flooding", in which all messages are forwarded by the devices acting as relays. This approach ensures that a message has multiple paths to arrive at its destination rather than following a specific route or going through a centralized router, meaning that there are no single points of failure, thus making the network very reliable [17]. To avoid a message to be infinitely forwarded through the network, the Bluetooth mesh includes some measures like the *message cache* and a Time To Live (TTL).

The *message cache* keeps track of messages that have been handled recently. Based on this cache, the device can filter out packets to determine whether it should retransmit a received message or if it should discard it immediately, avoiding unnecessary processing.

The Time To Live is a field included in the mesh packets, used to limit the number of times that a message can be relayed. Every time a device acting as a relay receives a message, will decrement the TTL value before forwarding it. This way, messages are forwarded by relays only until the TTL value reaches zero [17]. The TTL has a maximum default value of 127 but can be decreased according with the size of the network and number of relay nodes.

But despite those measures, the flooding-based approach can still cause a lot of redundant traffic, which has impact on the throughput and reliability of the network. Thus, a general recommendation is to limit the number of relays in the network.

In fact, a study on the performance of a large-scale mesh network conducted by Ericsson [23], concluded that, under certain conditions it might be possible to use the relay feature in only 1.5 percent of the total nodes in the network.

Such study was conducted on a building automation scenario with a total of 879 devices deployed in an office with an area of 2,000 square meters and takes into consideration two possible configurations on the relay nodes: *baseline configuration*, that only considers the TTL and message cache mechanisms, and an *enhanced network configuration*. Both configurations were tested with sparse relay deployment, using 12 relays uniformly distributed, and dense relay using 49 relays redundantly deployed.

Three traffic setups were considered in all the cases: a low-traffic case with aggregate application throughput of ~150 bps, a medium-traffic case with aggregate application throughput of ~1 Kbps, and a high-traffic case with aggregate application throughput of ~3 Kbps.

As a way of providing a better level of reliability in the network, the Bluetooth Mesh Profile specification also allows the repetition of messages at the network layer. So, when relaying a message, the node can be configured to send every network layer packet several times to the bearer layer below [18, p. 151]. Ericsson purposes, as an *enhanced network configuration*, that the source node should repeat each message three times, while relay nodes should only retransmit each message once, since the bottleneck of Bluetooth mesh *is often the first hop to inject the packet in the network*, according to their study.

Another option purposed is to add a random delay component when transmitting packets over all the different advertising channels, in order to decrease the probability of collisions on all channels simultaneously.

The performance metric of the mesh network used in this study was defined *as the ratio of transmitted packets that reach the end destination within 300ms*, which is considered a typical requirement for lighting applications.

Results showed that for low traffic and sparse relay deployment the *baseline network configuration* is enough for the Bluetooth mesh to perform satisfactorily, with 99.1 percent of messages successfully delivered. However, with high traffic, it was only

possible to achieve a satisfactory performance using the enhanced configuration. With the enhanced settings, the number of delivered messages goes up to 99.1 percent on the worst case (dense relay deployment and high traffic) and up to more than 99.9 percent on all the remaining cases.

The study concludes that the best performance is obtained when deploying six relays every 1,000sq m, which corresponds to about 1.5 percent of the total number of nodes. Figure 15 shows the percentage of messages successfully delivered for the various cases.

	Baseline			Enhanced		
	Low traffic	Medium traffic	High traffic	Low traffic	Medium traffic	High traffic
Sparse deployment	99.1%	95.4%	84.3%	>99.9%	>99.9%	>99.9%
Dense deployment	97.5%	88.7%	69.2%	>99.9%	>99.9%	>99.1%

Figure 15: Percentage of messages delivered within 300ms in the various cases of the study.

The number of relay-enabled devices in the network should be tuned according to network density, traffic volumes and network layout, so it might be worth to consider the results of this study when deciding on the number of relays in a Bluetooth mesh network.

In the nRF5 SDK for Mesh the relay function is enabled by default. This can be changed through a boolean variable called *m_relay_enable* in the **network_init()** function, which can be set either to true or false. The **network_init()** function can be found in the *network.c* file.

The relay setting can also be changed after the device has been programed, using the Configuration client model, which can configure a mesh device by communicating with a remote device's Configuration server model [24].

Through the function **config_client_relay_set()** of the Configuration client, it is possible to enable or disable the relay feature, as well as define the number times a relayed packet should be re-transmitted and the number of 10 milliseconds steps between each re-transmission.

3.1.6 Proxy node

There is a huge number of devices already in the market that do not support the Bluetooth mesh stack, but that support Bluetooth LE. To take advantage of this, the Bluetooth mesh specification defines a separate protocol for tunneling mesh messages over the Bluetooth low energy GATT protocol [25], the GATT Proxy Protocol [15].

This protocol allows Bluetooth low energy devices, like smartphones and tablets, to participate in the mesh network by establishing a GATT connection to a mesh device that has the proxy feature enabled.

Figure 16 shows an example of communication through a node with the proxy feature.

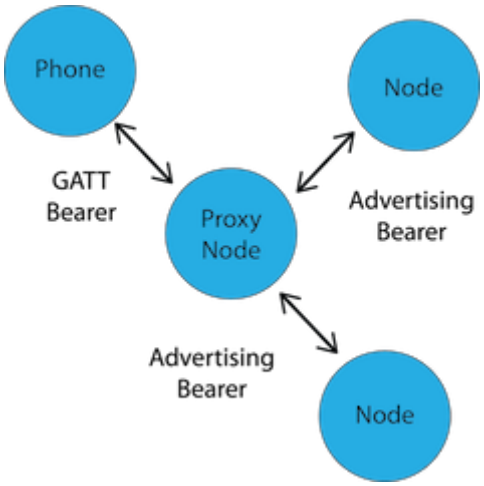

Figure 16: Example of communicating through a proxy node [26].

To make use of the nRF Mesh mobile application for provisioning the examples in the Nordic Mesh SDK, the proxy feature must be enabled in all the nodes. For this purpose, the light switch example has a separate project, for client and server, with the proxy functionality implemented. The applications developed in this thesis also make use of the proxy feature to allow provision with the nRF Mesh and, perhaps in a future work, to allow controlling the nodes from a smartphone.

3.1.7 Security

Security in Bluetooth mesh is mandatory. Thus, the specification implements a set of mechanisms to secure the network, individual applications and devices. One of those mechanisms is the Mesh Security Keys and Separation of Concerns.

Every Bluetooth mesh network message is secured using NetKeys and AppKeys to encrypt and authenticate messages and there is also a special type of key, the Device Key (DevKey), used in the provisioning process to secure communication between the Provisioner and the node [19, pp. 21-22].

A **NetKey** secures communication at the Network Layer and is shared across all nodes in the network. Having a given NetKey is what defines a node as member of a mesh network [21]. Thus, it is possible to subnet in Bluetooth mesh, by using a different NetKey for each subnet. This might be used, for example, to isolate specific physical areas, such as rooms in a hotel. Being in possession of the NetKey allows a node to decrypt and authenticate up to the Network Layer but it does not allow application data to be decrypted [19, p. 22].

An **AppKey** secures communication at the Access Layer and is shared across all nodes which participate in a given mesh Application [21]. It is generated and distributed during the provision process by the Provisioner. Application data for a specific application can only be decrypted by nodes which possess the right application key. For example, lights and light switches would have the lighting application's AppKey but not the AppKey for the HVAC system [19, p. 22].

For this reason, after provisioning a node with the nRF Mesh application, it is mandatory to add an AppKey to every element of the node (the application does not do it automatically). The android application has three AppKey by default, but more can be added in *Settings → Manage App Keys*.

4 RGB light control application

This is an application based on the light switch demo and the Simple OnOff model, designed to control one or more RGB LED lamps through a Bluetooth mesh network.

The application supports, without any additional software update, up to three RGB lamp (server nodes) and one switch (client node). Server nodes can be divided into two groups within the network, as shown in figure 17.

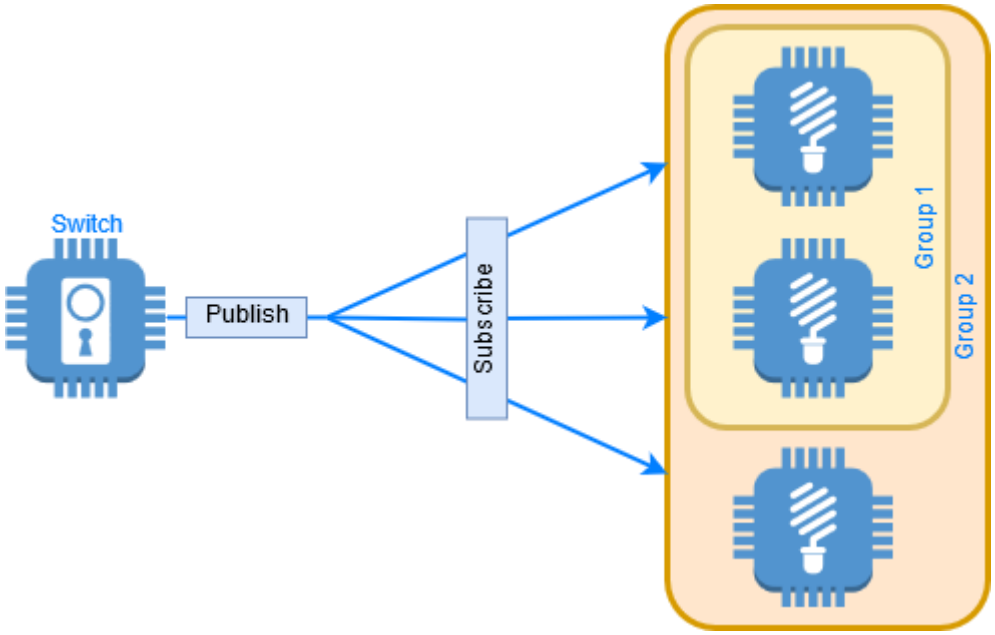

Figure 17: Illustration of the nodes on the light control application

The functionality of the server nodes is to drive the RGB LED light using PWM, controlled either locally with the hardware buttons or through the SET messages received from the client. Server nodes can also keep the status of the current RGB values in their flash memory so that they can return to the last status after a reboot or power off.

Client node is used to update the LED's colour and intensity on a single server node or a whole group, and to setup the status of the server on power up, it can be off, go to a default status or return to the last status before the server was rebooted or power off.

The application on the client node offers a command line-based user interface. Although this is enough to verify the functionalities of this light control system, the best user experience would be achieved using a tablet or smartphone.

Provisioning of the network is done using the nRF Mesh application for Android provided by Nordic Semiconductor. All code changes and functions mentioned in this chapter are shown in Appendix 2.

4.1 Integration with nRF5 SDK – Enabling drivers and libraries

Segger Embedded Studio expects the nRF5 SDK to be in a folder adjacent by default, i.e., it should be extracted to the same folder as nRF5 SDK for Mesh, otherwise, the path to nRF5 SDK should be set using the macro SDK_ROOT.

To do that, one must go to **Tools -> Options**, then **"Building"** and under **"Build"**, the setting **"Global macros"** should contain SDK_ROOT=<the path to the nRF5 SDK 15 instance>.

For example: *SDK_ROOT=C:\nRF5_SDK_15.0.0_a53641a*

But this is not enough to use the hardware drivers and other libraries from nRF5 SDK. To use PWM to control the colours of the RGB LED light, as it was done for this application, further steps are needed.

First, to enable the PWM driver in the light switch proxy server project, the path to the necessary include files must be set in the project properties. This is done by going to **Project -> Edit Options** and under **Preprocessor**, adding the bellow paths under the **User Include Directories**.

```
$(SDK_ROOT:../../../../nRF5_SDK_15.0.0_a53641a)/integration/nrfx/legacy
$(SDK_ROOT:../../../../nRF5_SDK_15.0.0_a53641a)/modules/nrfx/drivers/include
```

The C file that implements the PWM driver must also be added to the project. This is done with a right-click on the folder named nRF5 SDK in **Project Explorer**, then **"Add Existing File**" and selecting the nrfx_pwm.c file from the nRF5 SDK folder:

```
\nRF5_SDK_15.0.0_a53641a\modules\nrfx\drivers\src\nrfx_pwm.c
```

Finally, the PWM driver and at least one PWM instance must be enabled in the sdk_config.h file. This file is unique for each project and on the light switch proxy server example can be found at:

/nrf5_SDK_for_Mesh_v2.0.1_src/examples/light_switch/proxy_server/include/sdk_config.h.

To enable the driver and the instance, the value in the definition of NRFX_PWM_ENABLED and NRFX_PWM0_ENABLED must be updated from 0 to 1. Also, all the code related with the definition of the n**rf_drv_pwm - PWM peripheral driver** must be deleted. This is a legacy driver and will overwrite the definition of the **nrfx_pwm - PWM peripheral driver** done before.

This is all for the PWM driver. The integration of other drivers and libraries from the nRF5 SDK should be quite similar.

4.2 Modifying the Simple On-Off model

The Simple On-Off model supports messages that only can handle a single bit value, and for controlling an RGB led three values should be used instead. This implies changes on the message structure as well as in the handler functions that process them.

In the next the sections, the modifications needed to send three values over the Mesh network using the Simple On-Off model are described.

4.2.1 The messages structure

The message structure of the packets' payload sent through the mesh network is implemented in the header file **simple_on_off_common.h**, in the case of the Simple On-Off model.

In this file are defined also the company ID and the opcodes of the model, but for the RGB LED application, that information was left unchanged and only the message structure was modified to support the three new variables that correspond to the RGB values.

There are three structures defined one for Set, one for Set unreliable and one for Status messages. So, on those structures, the variable ***on_off*** was replaced by three new variables: ***red_dt_cycle***, ***green_dt_cycle*** and ***blue_dt_cycle***. For the Get message no special structure is needed, as it does not contain any information in the payload to be sent.

4.2.2 Modifying the Simple On-Off client

The client model can send three kinds of messages: a reliable (acknowledged) Set message, an unreliable (unacknowledged) Set message and a Get message. As explained before, the Get message does not contain any payload and thus the API responsible to send it did not require any modification.

However, the API used to send Set and Set Unreliable messages had to be adjusted to handle three integer variables instead of a single Boolean variable. This changes also apply to the handler that process the status reply to a Set reliable message.

To achieve this, the functions **simple_on_off_client_set()** and **simple_on_off_client_set_unreliable()** as well as the typedef **simple_on_off_status_cb_t** in the include file **simple_on_off_client.h**, were changed so that the Boolean variable ***on_off*** was replaced by the new variables ***red_dt_cycle***, ***green_dt_cycle*** and ***blue_dt_cycle***.

Those changes are also reflected in the simple On-Off client model's source. So, inside the file **simple_on_off_client.c**, the functions **simple_on_off_client_set()** and **simple_on_off_client_set_unreliable()** were updated so that the set messages sent from the client comply with the message structure defined in **simple_on_off_common.h**.

Also, the implementation of the status handler **handle_status_cb()** had to be changed to extract the three variables ***red_dt_cycle***, ***green_dt_cycle*** and ***blue_dt_cycle*** from the status message, instead of retrieving only one variable (the on off status).

4.2.3 Modifying the Simple On-Off server

In the Simple On-Off server model there are three opcode handlers to handle SIMPLE_ON_OFF_OPCODE_GET, SIMPLE_ON_OFF_OPCODE_SET, and SIMPLE_ON_OFF_OPCODE_SET_UNRELIABLE messages.

Each of these opcode handlers will call the corresponding user callback function from the context structure. This structure is defined in the include file **simple_on_off_server.h** and had to be modified to support the new message structure described in section 2.4.1. Also, in the same include file, the structure of the function

simple_on_off_server_status_publish() had to be modified as well. This function is called to send a status message every time there is a change in the server status, or in other words, it is used to 'inform' the client when the colour values of the LED have changed.

The callback functions mentioned above, are implemented in the **simple_on_off_server.c** file. such functions are responsible for processing the Set, Set Unreliable and Get messages, coming from the client.

The functions that handle the Set and Set Unreliable messages, **handle_set_cb** and **handle_set_unreliable_cb** respectively, were updated to extract the three different values from the message content. In the handler of the Set Unreliable messages the call to the reply status and the status publish were also removed, since acknowledgment is not needed for unreliable messages.

On the handler for the Get messages received from the client, the code was changed to handle the three variables and the callback **get_cb** was modified to make use of pointers to update more than one value when called from the main application.

The last two functions of the simple on-off server model that were updated are the **simple_on_off_server_status_publish()** and **reply_status()**. Both are quite similar, with the main difference being that the first one uses the **access_model_reply()** API to reply (acknowledge) a set message from the client, while the second one calls the **access_model_publish()** API, since it is used to publish a change in the server status.

4.3 The OnPowerUp model

The *OnPowerUp model* is a custom model inspired in the Generic OnPowerUp model [27, p. 31].It allows to define the behaviour of an element when powered up. The values for the server's state are defined on table 3.

Table 3: Description of the status supported by the OnPowerUp model.

Value	State	Description
0x00	Off	After being powered up, the element is in an off state.
0x01	Default	After being powered up, the element is in a state that uses default values.
0x02	Restore	After being powered up, the element restores the state it was in when powered down.

The implementation of this model was based on the Simple On-Off model, and thus, it has equivalent messages with similar functions and handlers. The main differences are, the value type supported by the messages, that are no longer a 1-bit value (Boolean) but instead an *uint8_t* to allow values from 0 to 2, and the status handler on the client side that now handles three different status.

The messages supported by the Store to Flash model are:

OnPowerUp Acknowledged Set – Reliable message sent from the client to define the status of the server after being powered up. In response to this message, the server should reply with a Store to Flash Status message.

OnPowerUp Get – Message used to get the status of the *store to flash* feature on the server.

OnPowerUp Unreliable – Unreliable message sent from the client to define the status of the server after being powered up. No reply message is needed from server side.

OnPowerUp Status – Reply message from the server in response to an Acknowledged Set message.

The new status types were defined in the include file **on_power_up_client.h** and are: ON_POWER_UP_STATUS_OFF, ON_POWER_UP_STATUS_RESTORE and ON_POWER_UP_STATUS_DEFAULT. The status handler implemented in the file **on_power_up_client.c**, had to modified accordingly to support those status.

The new opcodes for those messages were also defined in the respective structure declared the include file **on_power_up_common.h**.

In total, this model is implemented in 5 different files:

- on_power_up_client.h
- on_power_up_server.h
- on_power_up_common.h
- on_power_up_client.c
- on_power_up_client.c

The path to the include files must be set in the project properties and the C files added to the respective project, client or server, accordingly. This is done the same way as explained in section *2.2* for the driver files.

4.4 The server application

The server node application is responsible to drive the RGB light and for the flash operations. It has a single element running one instance of the modified Simple On-Off model and one instance of the On Power Up model, as shown in figure 18.

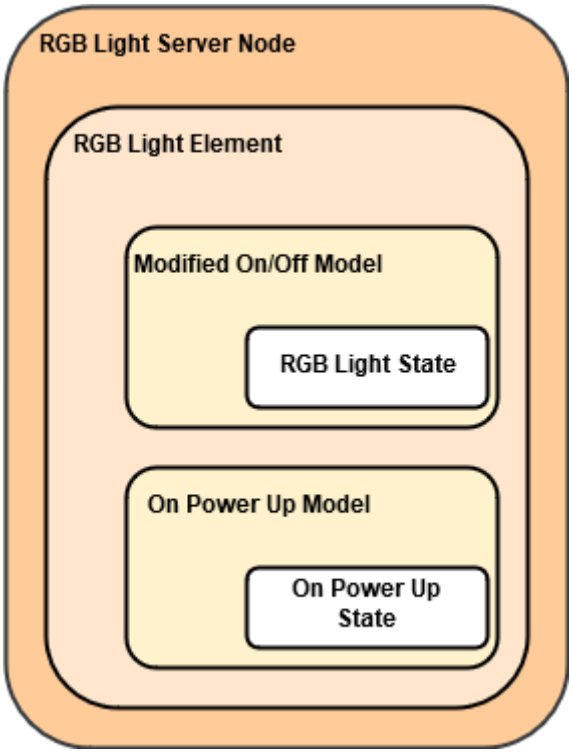

Figure 18: Structure of a server node running the RGB light control application.

The RGB light can also be controlled locally using the board's buttons and will publish those changes to the client, if it has a valid publish address configured.

Due to the limited number of buttons available on the DK, it's not possible to change the On Power Up status locally by hardware.
It is possible however, to connect the board to a computer and change the On Power Up status locally using a computer, to test its functionality.
The following sections show the changes implemented in the **main.c** that allow the application to work with the models and achieve its functionalities. This file can be found in the Application folder in the Project Explorer.

4.4.1 Implementation of PWM driver

The implementation of the PWM functionality in the server node project is based on the peripheral PWM driver example [28] from the nRF5 SDK.

First step was to add the include file **nrf_drv_pwm.h**. The path for the file also needs to be available in the Project options, as described in section 4.1.

The PWM driver supports multiple instances and each instance also to be declared separately. In this application the first instance was used by declaring NRF_DRV_PWM_INSTANCE(0). The value passed (zero) corresponds to the instance number enabled previously in the sdk_config.h.

The initialization and configuration of the driver for the given instance is done through the function **pwm_init()**. To avoid any conflict with interrupt priorities and to have the PWM driver ready before the node re-join the network, this function is called before the mesh initialization in the *main()* loop.

The driver supports 4 channels per instance, meaning that one instance can control up to 4 different signals, but in this application only 3 signals are needed, one for each colour (red, green and blue). So, the three first output pins were assigned to the GPIO pins 23, 24 and 25 of the development board (defined as RED_PIN, GREEN_PIN and BLUE_PIN), while the fourth one is assigned as NRFX_PWM_PIN_NOT_USED instead of a pin number to specify that its output it's not used, as described in **nrfx_pwm.h** include file.

In this function it's also possible to define the *top value* for the PWM duty cycle. In this case the top value was defined as 100, so the duty cycle will range from 0 to 100.

To update the PWM duty cycle values is used the function **pwm_update_duty_cycle()**. This function will update each channel based on the respective variable value **red_duty_cycle**, **green_duty_cycle** or **blue_duty_cycle**, that are used to hold the values of the three colours components of the LED.

The digital output pins on the development board are set to High by default, so in order to use a common cathode LED, the polarity must be inverted. Since the most significant bit (15) in the sequence of duty cycle values sets the polarity [29], it needs to be set to 1, in order to use the duty cycle value 0 as 0 Volts. This is done by using the bitwise OR operation as follow:

```
seq_values->channel_0 = duty_cycle | 0x8000;
```

To avoid the LED to turn on after the device is power on, the function **pwm_update_duty_cycle()** is called immediately after the PWM initialization to update the duty cycle to zero.

Finally, to control the LED from the server node board, the function **button_event_handler()** from the light switch example was modified to make use of the PWM update function. With those changes, it is possible to control the RGB LED using the hardware buttons on the board. Buttons 1, 2 and 3 control the colour red, green and blue respectively, while the button number 4 defines if the colour value should be increase or decrease.

To update the value of the variables holding the colour values before calling the PWM update function, is used another simple function named **update_value()**. This function is called every time one of the first three buttons is pushed. It first verifies if the current colour value should be increased or decreased, based on the value of a Boolean variable controlled by button 4, and then it increases or decreases the value by steps of 20, assuring always that the value will not be more than 100 or less than 0.

4.4.2 Adding flash manager

To make use of the OnPowerUp model functionality, the server application must be able to store the information about which status to select after being powered up and, in case of the status is defined as *Restore*, it must be also able to store the status of the LED.

Since in the Bluetooth mesh the nodes need to keep a persistent storage for the mesh data, the Mesh SDK already has a Flash Manager library [30] to handle the flash operations.

To save data in the flash it was defined a structure that consists of a 4-element array. First position of the array is used to save the OnPowerUp status and the remaining elements are used to store the status of the LED colour values.

To handle the write/read flash operations two functions were created, **write_to_flash()** and **read_from_flash()**. Inside the **write_to_flash()** there is function called **flash_manager_wait ()** that is used to prevent the function to terminate before the writing operation has finished and thus avoiding data corruption. However, that seems not to work and a 10ms delay was instead.

The very first time the device is powered up it should not refer to the status saved in the flash, as at that moment the flash contains just random data.

So, to define a OnPowerUp status for the next power cycles the Boolean variable ***m_device_provisioned*** is used. This variable is defined in the Nordic's implementation of the Bluetooth mesh and evaluates to false if the device is not provisioned, which is the case on first time the device is turn on.

So, the value 0, that corresponds to *status off*, is saved in the flash memory on the first start and will be the default value until it's changed through the OnPowerUp model. After the device has been provisioned, it will simply read from flash which status should be used and update the LED accordingly (if needed). The logic that controls this behaviour is shown in the flowchart of figure 19.

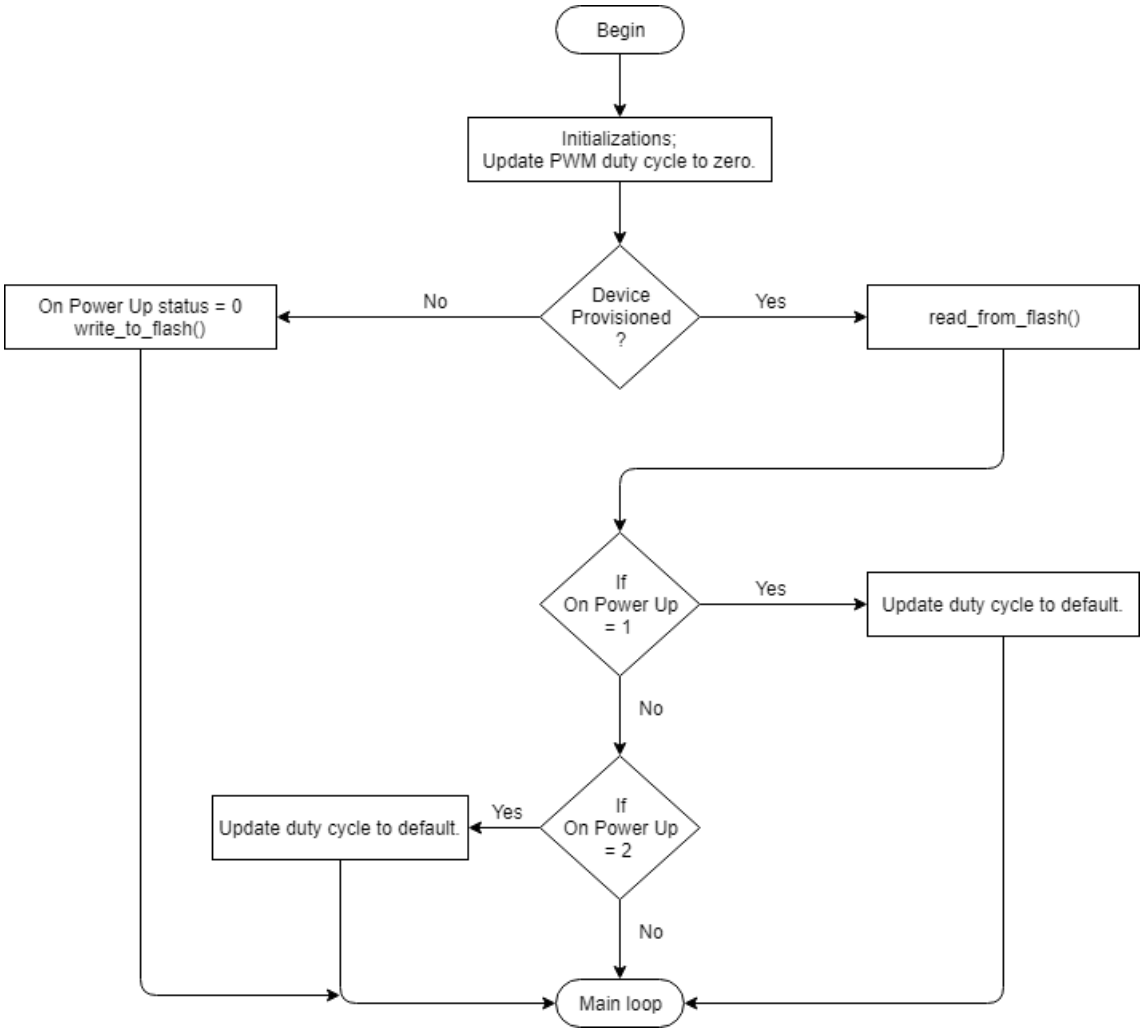

Figure 19: Behaviour of the server when powered on, regarding the OnPowerUp functionality.

To test this functionality, it is possible to connect the server to a computer and use the RTT terminal. The input ‘4’ sets the status to OFF, ‘5’ sets it to DEFAULT and ‘6’ to RESTORE.

4.4.3 Working with the models - server

To get the RGB light server application to work with the modifications on the simple On Off model, some adjustments had to be done to the original functions. It was also needed to create new functions for the application to interact with the On Power Up model.

So, the **on_off_server_get_cb()** function was modified to pass the current duty cycle values by reference to the callback function **get_cb()** on the simple on off model, instead of returning a single value as before.

On the other hand, the **on_off_server_set_cb()** function was updated, so that it receives the new values from the **set_cb()** function of the model and update the LED status by calling the **pwm_update_duty_cycle()** function, when the client sends a Set message.

The **on_off_server_set_cb()** function also checks the status of the On Power Up, if it's set to RESTORE, then the application must save the colour values in the flash. For that it uses a variable called *set_received* that when assigned to *true* triggers the **write_to_flash()** function.

To work with the On Power Up model, two new functions were created, the **on_power_up_get_cb()** and **on_power_up_set_cb()**, that handle the On Power Up Get and Set messages, respectively.

The **on_power_up_get_cb()** simply returns to the on_power_up_server_t **get_cb()** the current value of the on power up status server, while the function **on_power_up_set_cb()** handles the value received from the **set_cb()** function of the On Power Up model and updates the *on_power_up* variable on the server. It also assigns to true the *set_received* variable, triggering the function **write_to_flash()** that saves the new On Power Up status in the flash memory.

Finally, the **main()** function of the application has a loop that waits for events. Inside that loop an if statement was added to verify when the *set_received* becomes true. This works like an interrupt, if there is an update on the on power up status of the server, either due to a local event or a received Set message, the **write_to_flash()** function is called and the new status is saved in the flash memory. The same way, if a set message is received that changes the status of the LED and the OnPowerUp status is defined as *Restore*, the **write_to_flash()** is called to save the new LED colour values in the memory.

4.4.4 Adding a second model

To add the OnPowerUp model to the server node, two files must be updated in the respective project folder, the application file (**main.c**) and the **nrf_mesh_config_app.h**.

In the **nrf_mesh_config_app.h** the access model count and the access subscription list count must be incremented by one:

```
#define ACCESS_MODEL_COUNT (4)
#define ACCESS_SUBSCRIPTION_LIST_COUNT (2)
```

And in the application c file, a new instance of the static *on_power_up_server_t* was declared as **f_server** and the following lines were added to the function **models_init_cb()**:

```
f_server.get_cb = on_power_up_get_cb;
f_server.set_cb = on_power_up_set_cb;
ERROR_CHECK(on_power_up_server_init(&f_server, 0));
ERROR_CHECK(access_model_subscription_list_alloc(f_server.model_handle));
```

First two lines link the functions **get_cb()** and **set()** of the On Power Up model with the respective ones on the application level. The third line initializes the model on the element index 0. And finally, the fourth line allocates the memory to store the addresses' subscription list.

4.5 The client application

The client node application is used as switch to control the RGB light and to define the settings of the server node on power up. It has, besides the root element, five elements all running one instance of the modified Simple On-Off model and one instance of the On Power Up model each, as shown in figure 20.

First three client elements are meant for use with a unicast address, meaning that they only communicate with a single client, while the last two elements are intended for use with group addresses, so that they communicate with group 1 and group 2.
The unicast clients are implemented to send *acknowledged set messages* and the group clients use the *set unreliable message* format.

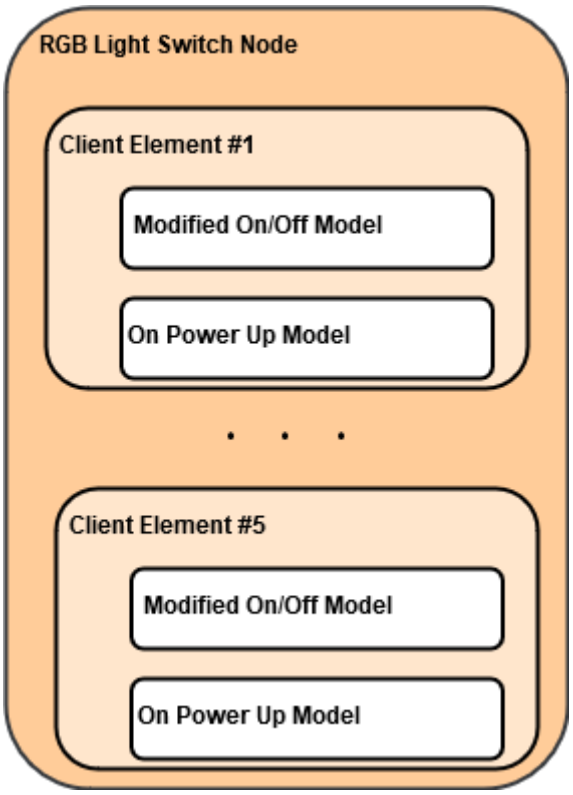

Figure 20: Structure of a client node running the RGB light control application.

Since the Client application is running in one of the DK boards, the interaction must be done through a command line-based interface, using the RTT terminal available in the Segger Embedded Studio.

4.5.1 Working with the models – client

In order to the user application to send GET and SET messages, it was needed to define handlers that call the right API functions of each model. So, on the client side, for each model there is a handler function for the SET messages, another for the GET messages and a function that handles the status reply from the server.

The **rgb_light_set_handler()** function is used to send a SET message to the light server. This function receives the index element ***i*** of the client that should send the message, and if the index corresponds to a unicast address client (0 to 2) then it calls the **simple_on_off_client_set()** to send a reliable (acknowledged) set message. Otherwise, if the index corresponds to a group address client it calls the **simple_on_off_client_set_unreliable()** to send a unreliable message to the group. To

verify the return of those functions, i.e., to verify if the message was sent with success or not, it's called the function **check_status()**.

The GET messages are sent by calling the function **rgb_light_get_handler()**. This function receives the index element ***i*** of the client that should send the message and call the function **simple_on_off_client_get()**. Then it verifies if the client corresponds to a unicast or group client, and only sends the GET message to the unicast addresses, since it's not suitable to send a get message to a group. Like the set handler, it uses the function **check_status()** to verify if the messages were sent with success.

To interact with the On Power Up server the application uses other two functions which are quite similar with the ones described above. To send SET messages is used the function **on_power_up_set_handler()**, while to send GET messages is used the **on_power_up_get_handler()**.

The **check_status()** function was implemented to reduce the amount of repeated code. It checks the return value passed through the variable *status* and compares it against the values defined in the include file **nrf_error.h**. A returned value of zero corresponds to NRF_SUCCESS, meaning that the message was sent successfully.

A server should reply to a reliable SET message by sending its status as acknowledgement. The reply status, on the application level, is handled by two distinct functions, one for each model. The same functions also handle the status message when the server publishes its status due to a local change. The status of each server is kept locally using a conjunct of arrays of five elements, one element for each client.

The status reply from a RGB light server is forward to the function **client_status_cb()**. This function basically updates the colour values on the respective array element, based on the ***server_index*** value and prints them in the command line. It also gives feedback in case the message does not get a reply from the server or if it gets an unknown status.

On the other hand, the status reply from an OnPowerUp server is forward to the function **f_client_status_cb()**. This function basically checks which status type was received (in the format *on_power_up_status_t*), updates the correspondent integer value in the respective array element, based on the ***server_index*** value, and prints the status

type in the command line. As in the previous function, it also gives feedback in case the message does not get a reply from the server or if it gets an unknown status.

4.5.2 Adding more elements and a second model

This application uses five client elements in total, three to communicate with the unicast addresses and two to communicate with the group addresses. Each of those client elements will run two models, the modified Simple On-Off and the On Power Up model.

Adding more client elements was done by increasing the CLIENT_MODEL_INSTANCE_COUNT on the **light_switch_example_common.h** to 5, as follow:

```
#define CLIENT_MODEL_INSTANCE_COUNT       (5)
```

Then, in the **nrf_mesh_config_app.h file,** the additional 5 addresses for the On Power Up model were added to the definition of ACCESS_MODEL_COUNT, as follow:

```
#define ACCESS_MODEL_COUNT (1 + /* Configuration server */ \
                            1 + /* Health server */ \
                            2 + /* Simple OnOff client (2 groups) */ \
                            3 + /* Simple OnOff client (3 unicast) */ \
                            2 + /* On Power Up client (2 groups) */ \
                            3   /* On Power Up client (3 unicast) */)
```

In the **main.c** a new array of type **on_power_up_client_t** was declared to hold the OnPowerUp clients:

```
static on_power_up_client_t f_clients[CLIENT_MODEL_INSTANCE_COUNT];
```

And finally, the following lines were added to the **models_init_cb()** function to initialize the second model on the client node:

```
f_clients[i].status_cb = f_client_status_cb;
f_clients[i].timeout_cb = client_publish_timeout_cb;
ERROR_CHECK(on_power_up_client_init(&f_clients[i], i + 1));
ERROR_CHECK(access_model_subscription_list_alloc(f_clients[i].model_handle));
```

The function **f_client_status_cb()** is described in the section 2.6.1, while the **client_publish_timeout_cb()** is exactly the same function used in the Simple On-Off model.

After those steps, the composition of the client node has 5 client instances with 2 models each.

4.5.3 Client interface

The interface is composed of different menus and sub menus, that allow the user to choose the client number to set/get the colour of the RGB light or to set/get the status of the OnPowerUp server status.

The input ranges from 0 to 9 in total. Each number correspond to a server (table 4).

Table 4: Input options for the client application interface.

Input index	**Address type**	**Model**
0, 1, 2	Unicast	Modified Simple On-Off
3, 4	Group	Modified Simple On-Off
5, 6, 7	Unicast	On Power Up
8, 9	Group	On Power Up

Since there are only five clients on the client node, the solution was to apply the modulus operator ('%') on the input number, like input%5, when handling messages to the servers of the On Power Up model.

The first level interface is implemented in the function **rtt_input_handler()**, shown in the flowchart of figure 21. This function is associated with an interrupt (1), so when a key is pressed in the RTT terminal the function is triggered and the input is checked if valid or not.

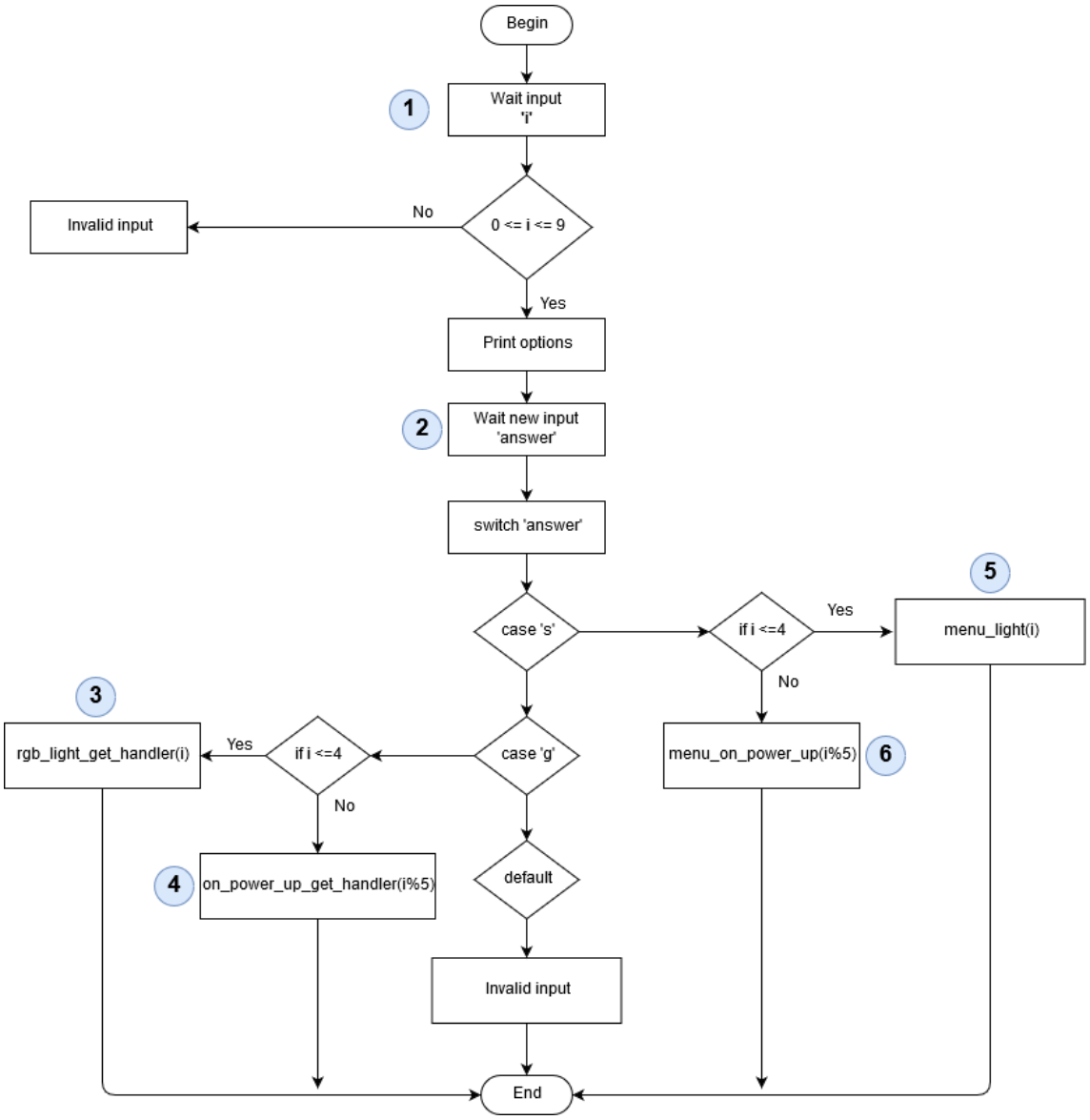

Figure 21: Flowchart describing the interface menus.

The second input (2) is read using the function **SEGGER_RTT_WaitKey()**, instead. This function basically holds the execution of the program until a key is pressed in the RTT terminal. The options printed in the command line terminal are ‘s’ to send a SET message and ‘g’ to send a GET message.

In case of GET messages, it will execute steps (3) or (4) depending on the input value, that corresponds to the functions **rgb_light_get_handler()** and **on_power_up_get_handler()**, respectively.

In the case of SET message, it will execute the **menu_light()** for the modified Simple On-Off model (5), or execute the **menu_on_power_up()** (6) for the OnPowerUp model.

The **menu_on_power_up()** function basically allow to update the value of the OnPowerUp status variable, *on_power_up[i]*, before calling the **on_power_up_set_handler()** that sends the Set message.

The **menu_light()**is a bit more complex. It allows to update the values of the colour components, *red_dt_cycle[i]*, *green_dt_cycle[i]* and *blue_dt_cycle[i]*, either choosing from a predefined set of colours or using a custom colour defined by the user. Any of the predefined colour options call the **defined_colour()** function to update the colour components, according with the colours available (Red, Green, Blue, Pink, Yellow, Cyan, White). The custom option calls the **custom_colour_menu()** that allows to update each colour component individually, using the keys '+' and '-' to respectively increase or decrease its value.

To reduce the amount of code in the **main.c** these two last functions were implemented in separate files, the **menu.h** and **menu.c**. Adding them to the project was done in similar way as described in section *4.4.1*.

4.6 Summary

This chapter described how to implement driver functionalities from the nRF5 SDK into a mesh application as well as how to modify the Simple On-Off model in order to implement a custom mesh application.

It was also showed how to build a new custom model and use it together with the flash support of the mesh SDK to control the state of a node after being powered up, as well as how to add more elements to a node and how to use more than one model in each element.

The result is an application capable of controlling one or more RGB lights over a Bluetooth mesh network, with the possibility of setting the light colour, individually or as a group, and the possibility of choosing what will be the state of the light after power up.

5 Sensor-driven Control System

In this section is described a possible implementation of a sensor-driven control system in a Bluetooth mesh network. Such control system, through its models, can control the power output of one or multiple devices based on the readings of a sensor device. It can be used in various applications like light brightness control using a light sensor or heating devices using a temperature sensor.

This system also introduces a new concept not mentioned in previous chapters, the Control Model. So far, it has been only described the Server and Client models, but the Bluetooth mesh specification also includes a Control Model [18, p. 23] which might contain server and client models in the same node, as well as a control logic that coordinate the interactions between other models that the Control Model connects to.

Figure 22 demonstrates the way nodes interact between them in this control system.

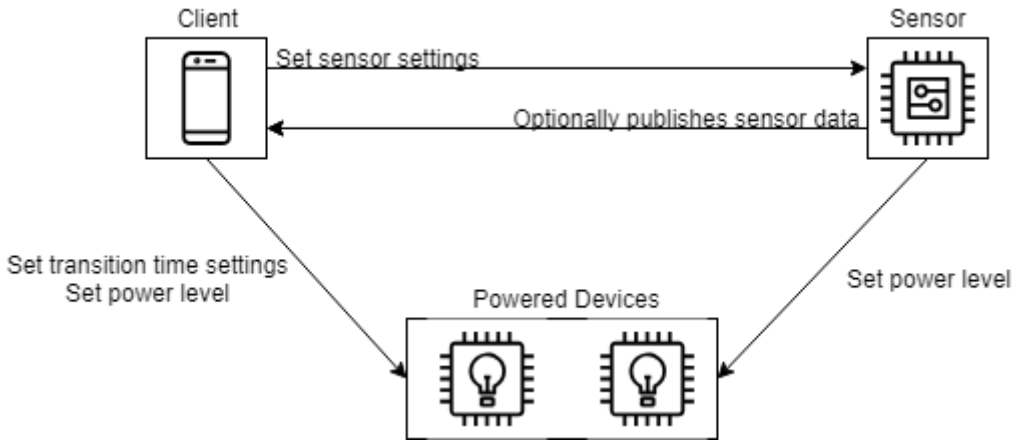

Figure 22: Interaction between nodes in the sensor-driven control system.

In this example the Client node is just another DK board running an application that allows to interact and configure the other nodes using a command line interface. Ideally, the Client node would be a smartphone or tablet running some application that could offer a graphical interface and perhaps allow the provisioning of the nodes as well.

All functions mentioned in this chapter are shown in Appendix 3.

5.1 Custom models

To implement this system three new models were created, the Sensor model, the Power Level model and the Transition Time model. All of them are based on the Simple On-Off model.

Those models aim to provide a generic support for working with sensors and to control power of different sorts of devices, enabling the nodes to exchange messages between them. While is at the application level that it should be specified what sensor type and what kind of device the node should control.

5.1.1 Sensor Model

The Sensor Model is a simple model inspired in the specification that defines the way of interfacing with sensors in Bluetooth mesh [18, p. 105]. It consists of two states, the Sensor Settings and the Sensor Data. The sensor settings have three different parameters, the Measurement Period, the Trigger Type and the Trigger Delta while the sensor data holds the measurement value itself.

The Sensor Measurement Period represents the period, in milliseconds, between two consecutive measurements.
The Status Trigger Type defines when the sensor should publish its status (Sensor Data). The value 0 on the trigger type, will cause the sensor to publish the measured value after every measurement, while the value 1 will set the sensor to publish only when a measured value differs from the previous measurement more than a defined percentage.
The Trigger Delta determines, in percentage, the minimum change (up or down) that triggers the publication of the Sensor Data.

To implement these functionalities, the sensor model supports the following messages:

Sensor Settings Set – Reliable message sent from the client to define the settings of the sensor. This message contains the measurement period, the trigger type and the trigger delta. In response, the server should send a Sensor Settings Status message.
Sensor Settings Get – Message used to retrieve the current settings of a sensor.

Sensor Settings Set Unacknowledged – Reliable message sent from the client to define the settings of the sensor. No reply is required from the server.
Sensor Settings Status – Message from server in reply to an acknowledged Sensor Settings Set.
Sensor Data Get – Message used to get the latest measured value of the sensor.
Sensor Status – Message that publishes the Sensor Data value, triggered by timer event or delta change.

5.1.2 Power Level Model

The Power Level model is a basic implementation of the Generic Power Level model [27, p. 73] and it's aimed to control the output power of an element.
It consists of two states, the Power Level Actual and the Power Level Default.
While the Power Level Actual is used to set the current power level of an element, the Power Level Default is used to determine the power level of an element when the device is powered on.

The messages supported by this model are:

Power Level Set – Reliable message from the client to define the value of the Power Level Actual status on the server. In response to this message, the server should reply with a Power Level Status message.
Power Level Get – Message that requests the current state of the Power Level Actual of the server.
Power Level Set Unacknowledged – Unreliable message from the client to define the value of the Power Level Actual status on the server. No reply status message is required.
Power Level Status – Reply message from the server in response to an Acknowledged Set message.

The messages for the Power Level Default are: **Power Default Get, Power Default Set, Power Default Set Unacknowledged** and **Power Default Status**.

The Power Level Default state shares the same message structure with the Power Level Actual state. In fact, besides the message structure, some handler functions are also shared between the two states.

5.1.3 Transition Time Model

The Transition Time model is a custom implementation based on the concept of the Generic Default Transition Time state [27, p. 29] and respective Generic model [27, p. 70]. It defines the time that an element shall take to change from the current state to a new state. It consists of two parameters, the number of transition steps and the step resolution.

The step resolution determines the time, in milliseconds, that each step should last, while the number of steps determines how many steps are used to change from one state to another. The transition time can be defined as:

$$Transition\ Time = Step\ Resolution \times Number\ of\ Steps$$

Such functionality is implemented by the following messages:

Transition Time Set – Reliable message from the client to define the number of steps and step resolution that a server should use for state transitions. In response to this message, the server should reply with a Transition Time Status message.
Transition Time Get – Message used to get the current number of steps and step resolution on the server.
Transition Time Set Unacknowledged – Unreliable message from the client to define the number of steps and step resolution. No reply is needed from the server side.
Transition Time Status – Message from server in reply to an acknowledged Transition Time Set.

5.2 Sensor node

The sensor device node contains a server instance of the Sensor model and a client instance of the Power Level model, as well as the *control logic* that decides whether the

sensor should publish its status or not, and consequently if the client should send a set message to the controlled devices or not. The node composition is shown in figure 23.

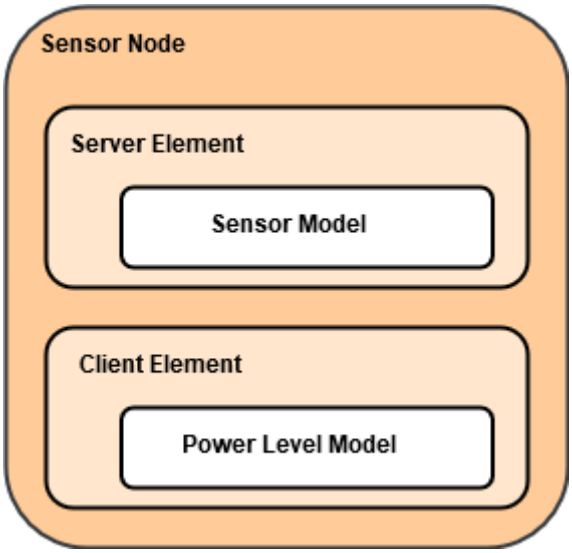

Figure 23: Composition of the sensor node.

The application on this node makes use of two peripheral drivers, the NRFX_TIMER and the NRFX_SAADC. The way to enable drivers is described on section 4.1 of this report.

The ADC is used to read the value from the *sensor* output. However, for demonstration, it was used a potentiometer to emulate the behaviour of the sensor, changing the voltage on the ADC pin like the output of the sensor would diverge according to changes on the surrounding environment. The timer is used to trigger an ADC reading every measurement period.

Two functions are needed to operate the timer, the **timeout_config()** and the **timeout_restart().** The first one is used to initialize the timer according with the default value. The second function is used to restart the timer with a new timeout value set by a Sensor Settings Set or Sensor Settings Set Unacknowledged message.

Both of the functions will call the function **timeout_event_handler()** after the timer has expired. This function uses the **nrf_drv_saadc_sample()** to consecutively read one sample from the ADC until the *sample buffer* is full. After that the function

saadc_callback() is triggered. This callback function calculates the average value of the samples in the buffer and update the variable *sensor_value* accordingly. At the end it sets the variable *sensor_read_finished* to true.

The variable *sensor_read_finished* works as a interrupter, and when it is assigned to true, it executes the function **status_publish_handler()** which is responsible for publishing the sensor value according to the trigger type. If the trigger type has the value 0, it will publish the sensor data regardless of its value, but if the trigger type is 1, it will calculate a delta (up and down) based on the trigger delta, and it will only publish the sensor data if there was a difference from the previous measurement, bigger than the delta value.

On both cases it uses the Power Level client to send a Power Level Set Unacknowledged message with the new power level based on the percentage of the sensor value, meaning that the client behaviour follows what is defined for the sensor settings. The Power Level client publishes to the group address 0xCAFE thus controlling both power control nodes at the same time.

5.3 Power control node

The power control node contains a single element with one server instance of the Power Level model and another instance of the Transition Time model. The role of this node is to control a device that can make use of a variable output power, based on the set messages receive from the sensor node, together with the settings of the transition time server. Its composition is shown in figure 24.

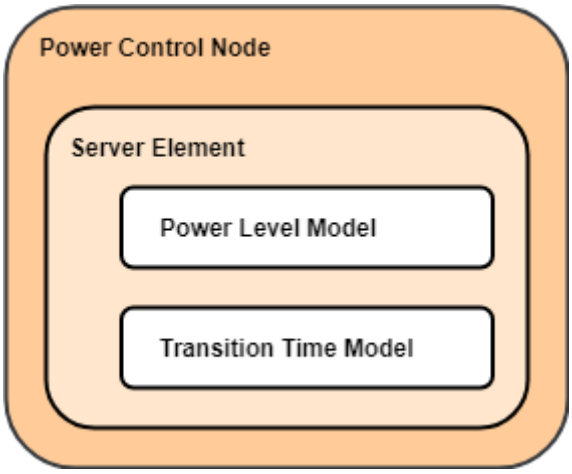

Figure 24: Composition of the power control node.

The application on this node uses the peripheral drivers NRFX_TIMER and NRFX_PWM. As for this example, the device controls the brightness of a LED making use of the PWM. The timer is used to implement the functionality of the Transition Time Model.

When the device receives a Power Level Set Unacknowledged from Power Level client on the sensor node, it initiates the actions needed to update the power level, that in this case will reflect in the *duty cycle* value of the PWM driver to be updated.

First action is to verify if the power level received is different from the previous value received, this is needed as the sensor node will cause the power level client to publish a set message regardless of the value, if the trigger type is defined as 0. If the power level received is different from the previous one, the application will stop any ongoing transition, if any, and initiate a new one by calling the function **set_power_actual_handler()**.

The function handler for the power transition calculates the value of each increment (delta) based on the difference between the actual power level and the new target level divided by the number of steps defined in the Transition Time server.

After this, the handler function runs either the timer functions **timeout_config()** or **timeout_restart()**, depending on either the timer needs to be set for the first time or it just needs to be restarted. Both functions run the timer with a timeout defined by the step resolution value of the Transition Time server.

Every time the timer expires, it executes one step by calling the function **timeout_event_handler()**, until it reaches the total number of steps. The timeout handler has the purpose of incrementing or decrement the actual power level by one *delta* value, calling the function **pwm_update_duty_cycle ()**. The process is repeated as many times as the total number of steps, if the number of steps is 1 the transition is immediate.

The power level default state is used to set the power level when the device is powered on. To make a good use of it, the support for flash memory must be added as it was done for the OnPowerUp model, in the RGB light control application.

5.4 Client node

The client node is the interface to interact with the other nodes in the system. It has a total of four elements, all of them containing one instance of the Power Level model and one instance of the Transition Time model. The first element also contains an instance of the Sensor model.

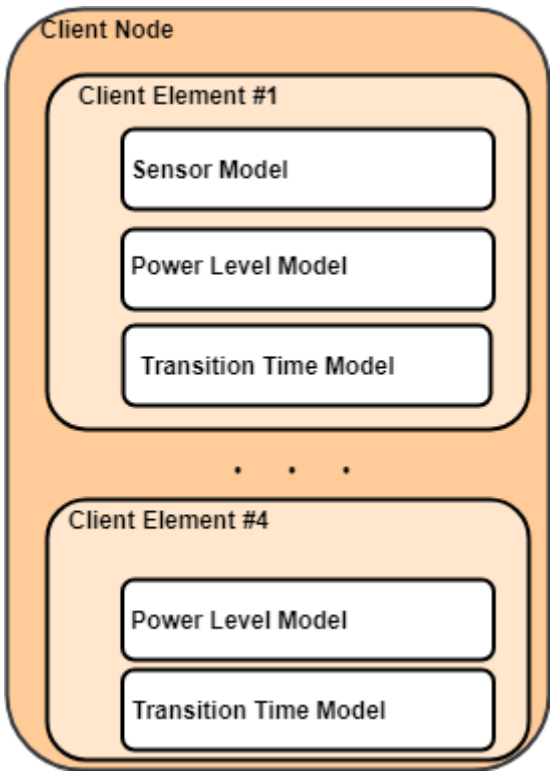

Figure 25: Composition of the client node.

It is used to configure the settings off the Sensor server model on the sensor node and optionally can subscribe to the Sensor Data status messages, allowing it to receive updates of the value read by the sensor.

It also controls the settings of the Transition Time Model server and can set the power level (actual and default) on the power control nodes. Contrary to the sensor node that only publishes to group the address, the client node can set the power level of the power control nodes individually or as a group.

Interface is command line-based and is triggered by typing one of the keys **‘s’**, **‘t’**, **‘p’** or **‘d’** in the RTT terminal.

The key ‘s’ will bring up the options for the sensor node: 0, 1 and 2. The numbers 0 and 1 will get the current sensor data and the current sensor settings, respectively. The number 2 will call the sensor settings handler where it is possible to set new settings for the sensor node.

The key ‘t’ gives access to the transition time settings. After typing this key, user will be asked for the server number, 0 and 1 for the first and second power control node, respectively, and 2 for both nodes (group address). Then user must choose between the options *get* or *set*. If *get* is chosen, it will simply send a *get settings* message to the respective server address; if *set* is chosen it will allow to update the number of steps and step resolution and send a *set settings* message.

Finally, the keys ‘p’ and ‘d’ will respectively open options for power level actual and power default value. Options are similar for both states, first the user has to choose server number and then *get* or *set* options. The option *get* sends a Power Level Get or Power Default Get message, depending which state user has chosen, while the *set* option will allow the user to update the power level actual or power level default, on one or both nodes simultaneously.

5.5 Summary

This chapter described an example on how to implement a sensor-driven control system in a Bluetooth mesh network using a combination of different models.

The models are intended to be general implementations that can be used in different solutions. For example, the sensor model is independent of the sensor type used at application level, it can be a temperature sensor or a light sensor, and it can use different type of interface ADC, SPI, I2C, etc. Likewise, the power level model can be used to control the brightness of a lamp or LED, as well as an HVAC equipment, for example.

To make full use of the power level default settings, the functionality of the power nodes can be extended with the OnPowerUp model from the chapter 4, besides the support for flash memory operations. The same way, to avoid the sensor node to override the power level actual when a user is setting it from the client node, the Simple On-Off model can be added to the sensor node to turn on or off the publishing of set power level messages.

Conclusions

Since its creation, Bluetooth technology has been adding new capabilities that made it a decisive player in the growth of IoT. Now, with the addition of mesh capability, it promises to continue that trend by extending its influence on new markets like Smart Building and Smart Home.

This thesis provided a good overview on the Bluetooth mesh technology, especially in concepts like publish/subscribe, group address, models and elements, provisioning, among others. Through the development of the light applications and the sensor-driven control system was possible to demonstrate/learn how to build applications to make use of existing models, as well as, how to build custom models and use multiple model instances to define the behaviour of a node.

Regarding the development platform, Nordic APIs and examples revealed to be rather more complex than other platforms like Arduino, for example. In addition, the SDK for Mesh did not fully support the mesh specification from the beginning, also, the updates with additional features were released sometimes more than a couple of months apart, resulting in increased difficulties in the development of the applications described in this thesis.

Future work

Regarding future work, developing a mobile application to implement the client functions, for both light control application and sensor-driven control system, is a must. A good starting point could be the nRF Mesh mobile application that is distributed as open source. This application can also be modified to present the name of each custom model, instead of presenting all as “vendor model”, to make it easier to configure the nodes.

Another interesting idea would be to replace some of the custom models by their equivalent Generic models that are described in the Bluetooth mesh specification, to verify their interoperability with other solutions from different manufactures, other than Nordic.

References

[1] Jaycon Systems, “Bluetooth Technology: What Has Changed Over The Years,” [Online]. Available: https://medium.com/jaycon-systems/bluetooth-technology-what-has-changed-over-the-years-385da7ec7154. [Accessed 10 12 2018].

[2] M. Woolley, “Bluetooth mesh networking: paving the way for smart lighting,” 2017. [Online]. Available: https://www.bluetooth.com/bluetooth-technology/topology-options/le-mesh/mesh-paving. [Accessed 20 12 2018].

[3] Silicon Laboratories Inc., “Bluetooth Mesh Solution Helps Cut Time to Market by Six Months,” 19 7 2017. [Online]. Available: http://powerpulse.net/bluetooth-mesh-solution-helps-cut-time-to-market-by-six-months/. [Accessed 21 12 2018].

[4] 2. b. K. K. July 18, “Introducing Bluetooth Mesh Networking,” Bluetooth SIG, 18 7 2017. [Online]. Available: https://blog.bluetooth.com/introducing-bluetooth-mesh-networking. [Accessed 12 11 2018].

[5] Nordic Semiconductor, “Product Brief,” [Online]. Available: https://www.semiconductorstore.com/cart/pc/viewPrd.asp?idproduct=61396#ProductInformation - Product Brief. [Accessed 14 02 2018].

[6] Nordic Semiconductor, “Quick Start Guide for the nRF5 SDK for Mesh,” 2018. [Online]. Available: http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.meshsdk.v2.1.1%2Fmd_doc_getting_started_mesh_quick_start.html. [Accessed 05 03 2018].

[7] Nordic Semiconductor, “nRF52 DK Product Brief,” [Online]. Available: https://www.semiconductorstore.com/cart/pc/viewPrd.asp?idproduct=61396#ProductInformation - nRF52 DK Product Brief. [Accessed 15 02 2018].

[8] Nordic Semiconductor, “SoftDevices,” 2018. [Online]. Available: http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.softdevices51%2Fdita%2Fnrf51%2Fsoftdevices.html. [Accessed 16 03 2018].

[9] Nordic Semiconductor, “nRF5 SDK,” 24 09 2018. [Online]. Available: http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52%2Fdita%2Fnrf52%2Fdevelopment%2Fnrf52_dev_kit%2Fhw_block_diag.html. [Accessed 06 10 2018].

[10] Nordic Semiconductor, “Simple OnOff model,” [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.meshsdk.v2.1.1%2Fmd_models_simple_on_off_README.html. [Accessed 09 04 2018].

[11] Nordic Semiconductor, “Light switch demo,” [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.meshsdk.v2.0.1%2Fmd_examples_light_switch_README.html. [Accessed 09 04 2018].

[12] SEGGER Microcontroller GmbH, “Embedded Studio — A Complete All-In-One Solution,” [Online]. Available: https://www.segger.com/products/development-tools/embedded-studio/?L=0. [Accessed 20 04 2018].

[13] SEGGER Microcontroller GmbH, “SEGGER Downloads,” [Online]. Available: https://www.segger.com/downloads/ . [Accessed 23 03 2018].

[14] M. Afaneh, “The Ultimate Bluetooth Mesh Tutorial (Part 1),” 03 09 2018. [Online]. Available: https://www.novelbits.io/bluetooth-mesh-tutorial-part-1. [Accessed 29 09 2018].

[15] Nordic Semiconductor, “Basic Bluetooth Mesh concepts,” 2018. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.meshsdk.v1.0.0%2Fmd_doc_introduction_basic_concepts.html. [Accessed 07 05 2018].

[16] Bluetooth Developer Relations Team, “The Fundamental Concepts of Bluetooth Mesh Networking, Part 2,” [Online]. Available: https://blog.bluetooth.com/the-fundamental-concepts-of-bluetooth-mesh-networking-part-2. [Accessed 14 05 2018].

[17] M. Woolley, “An Intro to Bluetooth Mesh Part 2,” 01 08 2018. [Online]. Available: https://blog.bluetooth.com/an-intro-to-bluetooth-mesh-part2. [Accessed 04 09 2018].

[18] Mesh Working Group, “Bluetooth Specification - Mesh Profile v1.0,” Bluetooth SIG, 2017.

[19] M. Woolley, “Bluetooth mesh networking - An Introduction for Developers,” [Online]. Available: https://www.bluetooth.com/bluetooth-technology/topology-options/le-mesh/mesh-tech. [Accessed 05 11 2018].

[20] K. Ren, “Provisioning a Bluetooth Mesh Network Part 1,” 18 09 2017. [Online]. Available: https://blog.bluetooth.com/provisioning-a-bluetooth-mesh-network-part-1. [Accessed 29 06 2018].

[21] “Bluetooth Mesh Glossary of Terms,” [Online]. Available: https://www.bluetooth.com/bluetooth-technology/topology-options/le-mesh/mesh-glossary. [Accessed 23 05 2018].

[22] Bluetooth Developer Relations Team, “The Fundamental Concepts of Bluetooth Mesh Networking Part 1,” 08 08 2017. [Online]. Available: http://blog.bluetooth.com/the-fundamental-concepts-of-bluetooth-mesh-networking-part-1. [Accessed 07 07 2018].

[23] Ericsson, “Bluetooth mesh networking,” 22 07 2017. [Online]. Available: https://www.ericsson.com/en/white-papers/bluetooth-mesh-networking. [Accessed 20 04 2018].

[24] Nordic Semiconductor, “Configuration client,” 29 01 2018. [Online]. Available: http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.meshsdk.v1.0.1%2Fgroup__CONFIG__CLIENT.html&anchor=gaaebf717a5ccb06b8c284a315d5d138c7. [Accessed 11 08 2018].

[25] Bluetooth SIG, “GATT Overview,” [Online]. Available: https://www.bluetooth.com/specifications/gatt/generic-attributes-overview. [Accessed 05 07 2018].

[26] M. Afaneh, “The Ultimate Bluetooth Mesh Tutorial (Part 2),” 10 09 2018. [Online]. Available: https://www.novelbits.io/bluetooth-mesh-tutorial-part-2/. [Accessed 15 11 2018].

[27] Mesh Working Group, “Bluetooth Specification - Mesh Model v1.0,” Bluetooth SIG, 2017.

[28] Nordic Semiconductor, “PWM Driver Example,” 03 04 2018. [Online]. Available:

https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.0.0%2Fpwm_hw_example.html&cp=4_0_1_4_5_22. [Accessed 12 06 2018].

[29] Nordic Semiconductor, “PWM HAL,” 24 11 2017. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v14.2.0%2Fstructnrf__pwm__sequence__t.html&cp=4_0_0_6_9_15_0_4_3&anchor=ad3244198df7ea3a206740dacd398db1e. [Accessed 03 19 2018].

[30] Nordic Semiconductor, “Flash manager,” 29 01 2018. [Online]. Available: http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.meshsdk.v1.0.1%2Fmd_doc_libraries_flash_manager.html&cp=4_1_0_4_0][http:/. [Accessed 07 07 2018].

Appendix 1 – Increasing number of elements on the server

main.c

```
#define SERVER_MODEL_INSTANCE_COUNT 2

//static simple_on_off_server_t m_server;
static simple_on_off_server_t m_server[1];

static bool on_off_server_get_cb(const simple_on_off_server_t * p_server)
{
    uint32_t index = p_server - &m_server[0];
    bool led_status;

        switch (index)
    {
        case 0:
        {
          led_status = hal_led_pin_get(LED_PIN_NUMBER);
          break;
        }

        case 1:
        {
          led_status = hal_led_pin_get(LED_PIN_NUMBER_1);
          break;
        }
    }
    return led_status;
}


static bool on_off_server_set_cb(const simple_on_off_server_t * p_server,
bool value)
{
    uint32_t index = p_server - &m_server[0];

    switch (index)
    {
        case 0:
        {
          __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Got SET command to %u\n",
value);
          hal_led_pin_set(LED_PIN_NUMBER, value);

          break;
        }
```

```
        case 1:
        {
          __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Got SET command to %u\n",
value);
          hal_led_pin_set(LED_PIN_NUMBER_1, value);

            break;
        }
    }
    return value;
}


static void models_init_cb(void)
{
    __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Initializing and adding models\n");

    for (uint32_t i = 0; i < SERVER_MODEL_INSTANCE_COUNT; ++i)
    {
      m_server[i].get_cb = on_off_server_get_cb;
      m_server[i].set_cb = on_off_server_set_cb;
      ERROR_CHECK(simple_on_off_server_init(&m_server[i], i));

ERROR_CHECK(access_model_subscription_list_alloc(m_server[i].model_handle));
      hal_led_mask_set(LEDS_MASK, false);
      hal_led_blink_ms(LEDS_MASK, LED_BLINK_INTERVAL_MS,
LED_BLINK_CNT_START);
    }
}
```

nrf_mesh_config_app.h

```
#define ACCESS_MODEL_COUNT (4)
#define ACCESS_ELEMENT_COUNT (2)
#define ACCESS_SUBSCRIPTION_LIST_COUNT (2)
```

Appendix 2 – Functions from the RGB light application

The messages structure:

```
typedef struct __attribute((packed))
{
    uint8_t red_dt_cycle; /**< Red value to set. */
    uint8_t green_dt_cycle; /**< Green value to set. */
    uint8_t blue_dt_cycle; /**< Blue value to set. */
    uint8_t tid;    /**< Transaction number. */
} simple_on_off_msg_set_t;

typedef struct __attribute((packed))
{
    uint8_t red_dt_cycle; /**< Red value to set. */
    uint8_t green_dt_cycle; /**< Green value to set. */
    uint8_t blue_dt_cycle; /**< Blue value to set. */
    uint8_t tid;    /**< Transaction number. */
} simple_on_off_msg_set_unreliable_t;

typedef struct __attribute((packed))
{
    uint8_t server_status;
    uint8_t current_red; /**< Current red state. */
    uint8_t current_green; /**< Current green state. */
    uint8_t current_blue; /**< Current blue state. */
} simple_on_off_msg_status_t;
```

Modifying the Simple On-Off client:

simple_on_off_client.h:

```
typedef void (*simple_on_off_status_cb_t)(const simple_on_off_client_t *
p_self, simple_on_off_status_t status, uint8_t server_red_status, uint8_t
server_green_status, uint8_t server_blue_status, uint16_t src);

uint32_t simple_on_off_client_set(simple_on_off_client_t * p_client, uint8_t
red_dt_cycle, uint8_t green_dt_cycle, uint8_t blue_dt_cycle);

uint32_t simple_on_off_client_set_unreliable(simple_on_off_client_t *
p_client, uint8_t red_dt_cycle, uint8_t green_dt_cycle, uint8_t
blue_dt_cycle, uint8_t repeats);
```

simple_on_off_client.c:

```
uint32_t simple_on_off_client_set(simple_on_off_client_t * p_client, uint8_t
red_dt_cycle, uint8_t green_dt_cycle, uint8_t blue_dt_cycle)
{
    if (p_client == NULL || p_client->status_cb == NULL)
    {
        return NRF_ERROR_NULL;
    }
    else if (p_client->state.reliable_transfer_active)
    {
        return NRF_ERROR_INVALID_STATE;
    }

    p_client->state.data.red_dt_cycle = red_dt_cycle;
    p_client->state.data.green_dt_cycle = green_dt_cycle;
    p_client->state.data.blue_dt_cycle = blue_dt_cycle;
    p_client->state.data.tid = m_tid++;

    uint32_t status = send_reliable_message(p_client,
                                            SIMPLE_ON_OFF_OPCODE_SET,
                                            (const uint8_t *)&p_client-
>state.data,
                                            sizeof(simple_on_off_msg_set_t));
    if (status == NRF_SUCCESS)
    {
        p_client->state.reliable_transfer_active = true;
    }
    return status;

}

uint32_t simple_on_off_client_set_unreliable(simple_on_off_client_t *
p_client, uint8_t red_dt_cycle, uint8_t green_dt_cycle, uint8_t
blue_dt_cycle, uint8_t repeats)
{
    simple_on_off_msg_set_unreliable_t set_unreliable;
    set_unreliable.red_dt_cycle = red_dt_cycle;
    set_unreliable.green_dt_cycle = green_dt_cycle;
    set_unreliable.blue_dt_cycle = blue_dt_cycle;
    set_unreliable.tid = m_tid++;
    access_message_tx_t message;
    message.opcode.opcode = SIMPLE_ON_OFF_OPCODE_SET_UNRELIABLE;
    message.opcode.company_id = SIMPLE_ON_OFF_COMPANY_ID;
    message.p_buffer = (const uint8_t*) &set_unreliable;
    message.length = sizeof(set_unreliable);
    message.force_segmented = false;
    message.transmic_size = NRF_MESH_TRANSMIC_SIZE_DEFAULT;

    uint32_t status = NRF_SUCCESS;
    for (uint8_t i = 0; i < repeats; ++i)
    {
        message.access_token = nrf_mesh_unique_token_get();
```

```
        status = access_model_publish(p_client->model_handle, &message);
        if (status != NRF_SUCCESS)
        {
            break;
        }
    }
    return status;
}

static void reliable_status_cb(access_model_handle_t model_handle,
                               void * p_args,
                               access_reliable_status_t status)
{
    simple_on_off_client_t * p_client = p_args;
    NRF_MESH_ASSERT(p_client->status_cb != NULL);

    p_client->state.reliable_transfer_active = false;
    switch (status)
    {
        case ACCESS_RELIABLE_TRANSFER_SUCCESS:
            /* Ignore */
            break;
        case ACCESS_RELIABLE_TRANSFER_TIMEOUT:
            p_client->status_cb(p_client,
SIMPLE_ON_OFF_STATUS_ERROR_NO_REPLY, NULL, NULL, NULL,
NRF_MESH_ADDR_UNASSIGNED);
            break;
        case ACCESS_RELIABLE_TRANSFER_CANCELLED:
            p_client->status_cb(p_client, SIMPLE_ON_OFF_STATUS_CANCELLED,
NULL, NULL, NULL,  NRF_MESH_ADDR_UNASSIGNED);
            break;
        default:
            /* Should not be possible. */
            NRF_MESH_ASSERT(false);
            break;
    }
}
```

Modifying the Simple On-Off server:

simple_on_off_server.h

```
typedef uint8_t (*simple_on_off_get_cb_t)(const simple_on_off_server_t *
p_self, uint8_t * current_red, uint8_t * current_green, uint8_t *
current_blue);

typedef uint8_t (*simple_on_off_set_cb_t)(const simple_on_off_server_t *
p_self, uint8_t red_dt_cycle, uint8_t green_dt_cycle, uint8_t blue_dt_cycle);

uint32_t simple_on_off_server_status_publish(simple_on_off_server_t *
p_server, uint8_t current_red_dt_cycle, uint8_t current_green_dt_cycle,
uint8_t current_blue_dt_cycle);
```

simple_on_off_server.c

```
static void handle_set_cb(access_model_handle_t handle, const
access_message_rx_t * p_message, void * p_args)
{
    simple_on_off_server_t * p_server = p_args;
    NRF_MESH_ASSERT(p_server->set_cb != NULL);

    uint8_t red_dt_cycle = (((simple_on_off_msg_set_t*) p_message->p_data)-
>red_dt_cycle);
    uint8_t green_dt_cycle = (((simple_on_off_msg_set_t*) p_message->p_data)-
>green_dt_cycle);
    uint8_t blue_dt_cycle = (((simple_on_off_msg_set_t*) p_message->p_data)-
>blue_dt_cycle);
    p_server->set_cb(p_server, red_dt_cycle, green_dt_cycle, blue_dt_cycle);
    reply_status(p_server, p_message, server_status,red_dt_cycle,
green_dt_cycle, blue_dt_cycle);
    (void) simple_on_off_server_status_publish(p_server, red_dt_cycle,
green_dt_cycle, blue_dt_cycle);
}
```

.

```
static void handle_set_unreliable_cb(access_model_handle_t handle, const
access_message_rx_t * p_message, void * p_args)
{
    simple_on_off_server_t * p_server = p_args;
    NRF_MESH_ASSERT(p_server->set_cb != NULL);
    uint8_t red_dt_cycle = (((simple_on_off_msg_set_t*) p_message->p_data)-
>red_dt_cycle);
    uint8_t green_dt_cycle = (((simple_on_off_msg_set_t*) p_message->p_data)-
>green_dt_cycle);
    uint8_t blue_dt_cycle = (((simple_on_off_msg_set_t*) p_message->p_data)-
>blue_dt_cycle);
    p_server->set_cb(p_server, red_dt_cycle, green_dt_cycle, blue_dt_cycle);
    (void)simple_on_off_server_status_publish(p_server, red_dt_cycle,
green_dt_cycle, blue_dt_cycle);  /* We don't care about status */
}

static void handle_get_cb(access_model_handle_t handle, const
access_message_rx_t * p_message, void * p_args)
{
    simple_on_off_server_t * p_server = p_args;
    NRF_MESH_ASSERT(p_server->get_cb != NULL);

    uint8_t current_red_dt_cycle;
    uint8_t current_green_dt_cycle;
    uint8_t current_blue_dt_cycle;
    p_server->get_cb(p_server, &current_red_dt_cycle,
&current_green_dt_cycle, &current_blue_dt_cycle);
    reply_status(p_server, p_message, server_status, current_red_dt_cycle,
current_green_dt_cycle, current_blue_dt_cycle);
}

uint32_t simple_on_off_server_status_publish(simple_on_off_server_t *
p_server, uint8_t current_red_dt_cycle, uint8_t current_green_dt_cycle,
uint8_t current_blue_dt_cycle)
{
    simple_on_off_msg_status_t status;
    status.server_status = server_status;
    status.current_red = current_red_dt_cycle;
    status.current_green = current_green_dt_cycle;
    status.current_blue = current_blue_dt_cycle;
    access_message_tx_t msg;
    msg.opcode.opcode = SIMPLE_ON_OFF_OPCODE_STATUS;
    msg.opcode.company_id = SIMPLE_ON_OFF_COMPANY_ID;
    msg.p_buffer = (const uint8_t *) &status;
    msg.length = sizeof(status);
    msg.force_segmented = false;
    msg.transmic_size = NRF_MESH_TRANSMIC_SIZE_DEFAULT;
      msg.access_token = nrf_mesh_unique_token_get();
    return access_model_publish(p_server->model_handle, &msg);
}
```

The OnPowerUp model

```
/** On Power Up status codes. */
typedef enum
{
    ON_POWER_UP_STATUS_OFF,
    ON_POWER_UP_STATUS_RESTORE,
    ON_POWER_UP_STATUS_DEFAULT,
    /** The server did not reply to a On Power Up Set/Get. */
    ON_POWER_UP_STATUS_ERROR_NO_REPLY,
    /** On Power Up Set/Get was cancelled. */
    ON_POWER_UP_STATUS_CANCELLED
} on_power_up_status_t;

static void handle_status_cb(access_model_handle_t handle, const
access_message_rx_t * p_message, void * p_args)
{
    on_power_up_client_t * p_client = p_args;
    NRF_MESH_ASSERT(p_client->status_cb != NULL);

    if (!is_valid_source(p_client, p_message))
    {
        return;
    }
    on_power_up_msg_status_t * p_status =
        (on_power_up_msg_status_t *) p_message->p_data;
    on_power_up_status_t on_power_up_status = (p_status->on_power_up_status);
    switch (on_power_up_status)
    {
      case '0':
          on_power_up_status = ON_POWER_UP_STATUS_OFF;
          break;
      case'1':
          on_power_up_status = ON_POWER_UP_STATUS_DEFAULT;
          break;
      case'2':
          on_power_up_status = ON_POWER_UP_STATUS_RESTORE;
          break;
      default:
          break;
    }
    p_client->status_cb(p_client, on_power_up_status, p_message-
>meta_data.src.value);
}
```

Implementation of PWM driver

```
#define RED_PIN       (23)
#define GREEN_PIN       (24)
#define BLUE_PIN       (25)

static nrf_drv_pwm_t m_pwm0 = NRF_DRV_PWM_INSTANCE(0);

static void pwm_init(void)
{
    nrf_drv_pwm_config_t const config0 =
    {
        .output_pins =
        {
            RED_PIN,              // channel 0
            GREEN_PIN,            // channel 1
            BLUE_PIN,             // channel 2
            NRF_DRV_PWM_PIN_NOT_USED,             // channel 3
        },
        .irq_priority = APP_IRQ_PRIORITY_LOWEST,
        .base_clock   = NRF_PWM_CLK_1MHz,
        .count_mode   = NRF_PWM_MODE_UP,
        .top_value    = 100,
        .load_mode    = NRF_PWM_LOAD_INDIVIDUAL,
        .step_mode    = NRF_PWM_STEP_AUTO
    };
    APP_ERROR_CHECK(nrf_drv_pwm_init(&m_pwm0, &config0, NULL));
}

void pwm_update_duty_cycle(uint8_t red_dt_cycle, uint8_t green_dt_cycle,
uint8_t blue_dt_cycle)
{
    {
        seq_values->channel_0 = red_dt_cycle | 0x8000;
        seq_values->channel_1 = green_dt_cycle | 0x8000;
        seq_values->channel_2 = blue_dt_cycle | 0x8000;
    }
    nrf_drv_pwm_simple_playback(&m_pwm0, &seq, 1, NRF_DRV_PWM_FLAG_LOOP);
}
```

```
static void button_event_handler(uint32_t button_number)
{
    __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Button %u pressed\n", button_number);
    switch (button_number)
    {
        case 0:
        {
            red_duty_cycle = update_value(red_duty_cycle);
            pwm_update_duty_cycle(red_duty_cycle, green_duty_cycle,
blue_duty_cycle);
            nrf_delay_ms(10);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Red value changed to %u \n",
red_duty_cycle);

            //(void)simple_on_off_server_status_publish(&m_server,
red_duty_cycle, green_duty_cycle, blue_duty_cycle);
            break;
        }

        case 1:
        {
            green_duty_cycle = update_value(green_duty_cycle);
            pwm_update_duty_cycle(red_duty_cycle, green_duty_cycle,
blue_duty_cycle);
            nrf_delay_ms(10);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Green value changed to %u
\n", green_duty_cycle);
            //(void)simple_on_off_server_status_publish(&m_server,
red_duty_cycle, green_duty_cycle, blue_duty_cycle);
            break;
        }

        case 2:
        {
            blue_duty_cycle = update_value(blue_duty_cycle);
            pwm_update_duty_cycle(red_duty_cycle, green_duty_cycle,
blue_duty_cycle);
            nrf_delay_ms(10);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Blue value changed to %u \n",
blue_duty_cycle);
            //(void)simple_on_off_server_status_publish(&m_server,
red_duty_cycle, green_duty_cycle, blue_duty_cycle);

        }

        case 3:
        {
            plus = !plus;
            hal_led_blink_ms(LED_PIN_MASK, LED_BLINK_INTERVAL_MS,
LED_BLINK_CNT_START);
            break;
        }
```

```
        default:
            break;
    }

}
```

Adding flash manager

```
typedef struct
 {
   uint32_t data[4];
 } custom_data_format_t;

static int write_to_flash(void)
{
    fm_entry_t * p_entry =
    flash_manager_entry_alloc(&m_custom_data_flash_manager,
    FLASH_CUSTOM_DATA_GROUP_ELEMENT, sizeof(custom_data_format_t));
    if (p_entry == NULL)
    {
      return NRF_ERROR_BUSY;
    }
      else
    {
       custom_data_format_t * p_custom_data = (custom_data_format_t *)
       p_entry->data;
      p_custom_data->data[0] = on_power_up;
      p_custom_data->data[1] = red_duty_cycle;
      p_custom_data->data[2] = green_duty_cycle;
      p_custom_data->data[3] = blue_duty_cycle;

      flash_manager_entry_commit(p_entry);
       __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "write:%u, %u, %u, %u\n",p_entry-
       >data[0], p_entry->data[1],p_entry->data[2], p_entry->data[3]);
    }
    flash_manager_wait();
}
```

```
static void read_from_flash()
{
    const fm_entry_t * p_read_raw =
    flash_manager_entry_get(&m_custom_data_flash_manager,
    FLASH_CUSTOM_DATA_GROUP_ELEMENT);

    const custom_data_format_t * p_read_data = (const custom_data_format_t *)
    p_read_raw->data;
    __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "read:%u, %u, %u, %u\n",p_read_data-
    >data[0], p_read_data->data[1], p_read_data->data[2], p_read_data-
    >data[3]);

    on_power_up = p_read_data->data[0];
    stored_red_value = p_read_data->data[1];
    stored_green_value = p_read_data->data[2];
    stored_blue_value = p_read_data->data[3];
}
```

main()

```
    flash_manager_config_t custom_data_manager_config;
    custom_data_manager_config.write_complete_cb = NULL;
    custom_data_manager_config.invalidate_complete_cb = NULL;
    custom_data_manager_config.remove_complete_cb = NULL;
    custom_data_manager_config.min_available_space = WORD_SIZE;

    custom_data_manager_config.p_area = (const flash_manager_page_t *)
    (((const uint8_t *) dsm_flash_area_get()) - (ACCESS_FLASH_PAGE_COUNT *
    PAGE_SIZE) - (NET_FLASH_PAGE_COUNT * PAGE_SIZE) );

    custom_data_manager_config.page_count = CUSTOM_DATA_FLASH_PAGE_COUNT;

    ret_code = flash_manager_add(&m_custom_data_flash_manager,
    &custom_data_manager_config);

    if (NRF_SUCCESS != ret_code) {
    __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Flash error: no memory\n",ret_code)
    }
```

Working with the models – server:

```
static void on_off_server_get_cb(const simple_on_off_server_t * p_server,
uint8_t * current_red_dt_cycle, uint8_t * current_green_dt_cycle, uint8_t *
current_blue_dt_cycle)
{

    * current_red_dt_cycle = red_duty_cycle;
    * current_green_dt_cycle = green_duty_cycle;
    * current_blue_dt_cycle = blue_duty_cycle;
}


static uint8_t on_off_server_set_cb(const simple_on_off_server_t * p_server,
uint8_t red_dt_cycle, uint8_t green_dt_cycle, uint8_t blue_dt_cycle)
{
    __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Red: %u, Green: %u, Blue: %u\n",
red_dt_cycle, green_dt_cycle, blue_dt_cycle);
    red_duty_cycle = red_dt_cycle;
    green_duty_cycle = green_dt_cycle;
    blue_duty_cycle = blue_dt_cycle;
    pwm_update_duty_cycle(red_duty_cycle, green_duty_cycle, blue_duty_cycle);
    nrf_delay_ms(10);
    if (on_power_up == 2)
    {
        set_received = true;
    }
}



static uint8_t on_power_up_get_cb(const on_power_up_server_t * p_server)
{
    return on_power_up;
}
static uint8_t on_power_up_set_cb(const simple_on_off_server_t * p_server,
uint8_t value)
{
    __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Got SET flash command to %u\n",
value);

    on_power_up = value;

    set_received = true;
    hal_led_blink_ms(LEDS_MASK, LED_BLINK_INTERVAL_MS, LED_BLINK_CNT_START);
    return value;
}
```

Working with the models – client:

```
static void rgb_light_set_handler(uint32_t i)
{
    uint32_t status = NRF_SUCCESS;

    if (i<=2)
    {
        status = simple_on_off_client_set(&m_clients[i],
                        red_dt_cycle[i], green_dt_cycle[i], blue_dt_cycle[i]);
    }
    else
    {
        status = simple_on_off_client_set_unreliable(&m_clients[i],
                                            red_dt_cycle[i], green_dt_cycle[i],
             blue_dt_cycle[i], GROUP_MSG_REPEAT_COUNT);
    }
    status = check_status(status, i);
    if (status == NRF_SUCCESS)
    {
        __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "RGB LED: SET command sent to
server %u.\n", i);
    }
}

static void rgb_light_get_handler(uint32_t i)
{
    uint32_t status = NRF_SUCCESS;

    if (i<=2)
    {
        simple_on_off_client_get(&m_clients[i]);

            status = check_status(status, i);
            if (status == NRF_SUCCESS)
            {
                __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "RGB LED: GET command sent
to server %u.\n", i);
            }
    }
    else
    {
        __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "The index %u is for a group
address. Action ignored.\n", i);
    }
}
```

```
static void on_power_up_set_handler(uint32_t i)
{
    uint32_t status = NRF_SUCCESS;

    if (i<=2) {
        status = on_power_up_client_set(&f_clients[i], on_power_up[i]);
    }
    Else {
        status = on_power_up_client_set_unreliable(&f_clients[i],
on_power_up[i], GROUP_MSG_REPEAT_COUNT);
    }

    status = check_status(status, i);
    if (status == NRF_SUCCESS) {
        __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "On Power Up: SET command to
server %u.\n", i);
   }
}


static void on_power_up_get_handler(uint32_t i)
{
    uint32_t status = NRF_SUCCESS;

    if (i<=2)
    {
        on_power_up_client_get(&f_clients[i]);
    }

    status = check_status(status, i);
    if (status == NRF_SUCCESS)
    {
        __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "On Power Up: GET command to
server %u.\n", i);
    }

}
```

```
static uint32_t check_status(uint32_t status, uint8_t i)
{
    switch (status)
    {
        case NRF_SUCCESS:
            break;

        case NRF_ERROR_NO_MEM:
        case NRF_ERROR_BUSY:
        case NRF_ERROR_INVALID_STATE:
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Cannot send - client %u is
busy\n", i);
            hal_led_blink_ms(LEDS_MASK, LED_BLINK_SHORT_INTERVAL_MS,
LED_BLINK_CNT_NO_REPLY);
            break;

        case NRF_ERROR_INVALID_PARAM:
            /* Publication not enabled for this client. One (or more) of the
following is wrong:
             * - An application key is missing, or there is no application
key bound to the model
             * - The client does not have its publication state set
             *
             * It is the provisioner that adds an application key, binds it
to the model and sets
             * the model's publication state.
             */
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Publication not configured
for client %u\n", i);
            break;

        default:
            ERROR_CHECK(status);
            break;
    }

    return status;
}
```

```
static void client_status_cb(const simple_on_off_client_t * p_self,
simple_on_off_status_t status, uint8_t server_red_status, uint8_t
server_green_status, uint8_t server_blue_status, uint16_t src)
{
    uint32_t server_index = server_index_get(p_self);

    red_dt_cycle [server_index] = server_red_status;
    green_dt_cycle [server_index] = server_green_status;
    blue_dt_cycle [server_index] = server_blue_status;

    switch (status)
    {
        case SIMPLE_ON_OFF_STATUS_ON:
            __LOG(LOG_SRC_APP, LOG_LEVEL_ERROR, "Server %u status received:
\n", server_index);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "RED: %u\n", red_dt_cycle
[server_index]);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "GREEN: %u\n", green_dt_cycle
[server_index]);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "BLUE: %u\n", blue_dt_cycle
[server_index]);
            break;

        case SIMPLE_ON_OFF_STATUS_ERROR_NO_REPLY:
            hal_led_blink_ms(LEDS_MASK, LED_BLINK_SHORT_INTERVAL_MS,
LED_BLINK_CNT_NO_REPLY);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "No reply from On Power Up
server %u\n", server_index);
            break;

        case SIMPLE_ON_OFF_STATUS_CANCELLED:
        __LOG(LOG_SRC_APP, LOG_LEVEL_WARN, "Message to server %u
cancelled\n", server_index);
        default:
            __LOG(LOG_SRC_APP, LOG_LEVEL_ERROR, "Unknown status \n");
            break;
    }
}
```

```
static void f_client_status_cb(const on_power_up_client_t * p_self,
on_power_up_status_t status, uint16_t src)
{
    uint32_t server_index =f_server_index_get(p_self);

    switch (status)
    {
        case ON_POWER_UP_STATUS_OFF:
            on_power_up [server_index] = 0;
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "On Power Up server %u status:
OFF\n", server_index);
            break;

        case ON_POWER_UP_STATUS_DEFAULT:
            on_power_up [server_index] = 1;
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "On Power Up server %u status:
DEFAULT\n", server_index);
            break;

        case ON_POWER_UP_STATUS_RESTORE:
            on_power_up [server_index] = 2;
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "On Power Up server %u status:
RESTORE\n", server_index);
            break;

        case ON_POWER_UP_STATUS_ERROR_NO_REPLY:
            hal_led_blink_ms(LEDS_MASK, LED_BLINK_SHORT_INTERVAL_MS,
LED_BLINK_CNT_NO_REPLY);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "No reply from On Power Up
server %u\n", server_index);
            break;

        case ON_POWER_UP_STATUS_CANCELLED:
            __LOG(LOG_SRC_APP, LOG_LEVEL_WARN, "Message to server %u
cancelled\n", server_index);
            break;
        default:
            __LOG(LOG_SRC_APP, LOG_LEVEL_ERROR, "Unknown status \n");
            break;
    }
}
```

Appendix 3 – Functions from the sensor-driven control system

Sensor node

```
void timeout_config(uint32_t milisecond)
{
    uint32_t time_ms = milisecond; //Time(in miliseconds) between consecutive
compare events.
    uint32_t time_ticks;
    uint32_t err_code = NRF_SUCCESS;

    nrf_drv_timer_config_t timer_cfg = NRF_DRV_TIMER_DEFAULT_CONFIG;
    err_code = nrf_drv_timer_init(&TIMER_PROVISION, &timer_cfg,
timeout_event_handler);
    APP_ERROR_CHECK(err_code);

    time_ticks = nrf_drv_timer_ms_to_ticks(&TIMER_PROVISION, time_ms);

    nrf_drv_timer_extended_compare(&TIMER_PROVISION,
                    NRF_TIMER_CC_CHANNEL1,
                    time_ticks,
                    NRF_TIMER_SHORT_COMPARE1_CLEAR_MASK,
                    true);

     nrf_drv_timer_enable(&TIMER_PROVISION);
}


void timeout_restart(uint32_t milisecond)
{
    uint32_t time_ms = milisecond; //Time(in milliseconds) between
consecutive compare events.
    uint32_t time_ticks;

    time_ticks = nrf_drv_timer_ms_to_ticks(&TIMER_PROVISION, time_ms);

    nrf_drv_timer_extended_compare(&TIMER_PROVISION,
                    NRF_TIMER_CC_CHANNEL1,
                    time_ticks,
                    NRF_TIMER_SHORT_COMPARE1_CLEAR_MASK,
                    true);

     nrf_drv_timer_enable(&TIMER_PROVISION);
}

…
```

```
void timeout_event_handler(nrf_timer_event_t event_type, void* p_context)
{
    uint8_t value = !hal_led_pin_get(BSP_LED_1);

    switch (event_type)
    {
        case NRF_TIMER_EVENT_COMPARE1:
             hal_led_pin_set(BSP_LED_1, value);
             __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Timer event\n");
            for (uint8_t i=0; i < SAMPLES_IN_BUFFER; i++)
            {
              nrf_drv_saadc_sample();
             __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Reading\n");
            }
            break;
        default:
            //Do nothing.
            break;
    }
}


void saadc_callback(nrf_drv_saadc_evt_t const * p_event)
{
    if (p_event->type == NRF_DRV_SAADC_EVT_DONE)
    {
        ret_code_t err_code;
        uint32_t average = 0;

        err_code = nrf_drv_saadc_buffer_convert(p_event->data.done.p_buffer,
SAMPLES_IN_BUFFER);
        ERROR_CHECK(err_code);
        for (uint8_t i = 0; i < SAMPLES_IN_BUFFER; i++)
        {
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO,"%d\n", p_event-
>data.done.p_buffer[i]);
            if (p_event->data.done.p_buffer[i] >= 0)
            {
                average += p_event->data.done.p_buffer[i];
            }
            else
            {
                average += 0;
            }
        }
        sensor_value = (float)average/SAMPLES_IN_BUFFER;

        sensor_read_finished = true;
    }
}
```

```
void status_publish_handler()
{
    bool power_publish = false;

    __LOG(LOG_SRC_APP, LOG_LEVEL_INFO,"Value read from sensor: %d\n",
(int)sensor_value); //convert to int just to print the number (needs a
workaround to print float)
    if (trigger_type == 0)
    {
        sensor_status_publish(&m_server, sensor_value);
        __LOG(LOG_SRC_APP, LOG_LEVEL_INFO,"Value sent (due to expired timer).
\n");
        power_publish = true;
    }
    else
    {
        float delta_down = (float)previous_value * (1 - (trigger_delta /
100.00));
        float delta_up = (float)previous_value * (1 + (trigger_delta /
100.00));

        __LOG(LOG_SRC_APP, LOG_LEVEL_INFO,"Lower delta: %d\n",
(int)delta_down); //convert to int just to print the number (needs a
workaround to print float)
        __LOG(LOG_SRC_APP, LOG_LEVEL_INFO,"Upper delta: %d\n",
(int)delta_up); //convert to int just to print the number (needs a workaround
to print float)

        if ((sensor_value < delta_down) || (sensor_value > delta_up))
        {
            previous_value = sensor_value;
            sensor_status_publish(&m_server, sensor_value);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO,"Value sent (due to delta
change). \n");
            power_publish = true;
        }
    }
    if (power_publish)
    {
        uint16_t power_level = (sensor_value * 100) / 800;
        __LOG(LOG_SRC_APP, LOG_LEVEL_INFO,"Power level: %d\n", power_level);

        uint32_t status = NRF_SUCCESS;

        status = power_actual_client_set_unreliable(&m_client, power_level,
GROUP_MSG_REPEAT_COUNT);

            switch (status)
        {
            case NRF_SUCCESS:
                break;
```

```
            case NRF_ERROR_NO_MEM:
            case NRF_ERROR_BUSY:
            case NRF_ERROR_INVALID_STATE:
                __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Cannot send - power level
client is busy\n");
                hal_led_blink_ms(LEDS_MASK, LED_BLINK_SHORT_INTERVAL_MS,
LED_BLINK_CNT_NO_REPLY);
                break;

            case NRF_ERROR_INVALID_PARAM:
                __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Publication not
configured for power client client\n");
                break;

            default:
                ERROR_CHECK(status);
                break;
        }
    }
    sensor_read_finished = false;
}
```

Power control node

```
void set_power_actual_handler()
{
    float temp = abs(duty_cycle_actual - duty_cycle_new) /
(float)number_of_steps;

    if( temp < 0.5)
    {
        delta = 1;
    }
    else
    {
        delta = round(temp);
    }
    power_last = duty_cycle_new;

    if (timer_on)
    {
        timeout_restart(step_resolution);
    }
    else
    {
        timeout_config(step_resolution); //If the timer is not running yet,
starts it for the first time
    }
    set_power_actual = false;
}
```

```
void timeout_event_handler(nrf_timer_event_t event_type, void* p_context)
{
    uint8_t value = !hal_led_pin_get(BSP_LED_1); //Signals that an update
operation is ongoing
    bool target_value = false;

    switch (event_type)
    {
        case NRF_TIMER_EVENT_COMPARE1:
             hal_led_pin_set(BSP_LED_1, value);
             if (duty_cycle_actual <= duty_cycle_new) //If target (new value)
is bigger that actual value, means we want to increase actual value.
             {
                duty_cycle_actual = duty_cycle_actual + delta;
                if (duty_cycle_actual == duty_cycle_new) //Means that we
reached our target value before the max number of steps. We can stop
operation now.
                {
                    target_value = true;
                }
             }
             else // If not, if target (new value) is smaller that actual
value, means we want to decrease actual value.
             {
                duty_cycle_actual = duty_cycle_actual - delta;
                if (duty_cycle_actual == duty_cycle_new) //Means that we
reached our target value before the max number of steps. We can stop
operation now.
                {
                    target_value = true;
                }
             }
             pwm_update_duty_cycle(duty_cycle_actual);
             __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Power updated to: %d. Delta:
%d\n", duty_cycle_actual, delta);
             step_counter += 1;
             __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Timer event %d\n",
step_counter);
            break;

        default:
            //Do nothing.
            break;
    }
    if (step_counter >= number_of_steps || target_value) //If we reach the
target value or the max number of steps
    {
        step_counter = 0;
        nrf_drv_timer_disable(&TIMER_PROVISION);
        /*In case we don't reach the exact target value (due to the error
introduced when rounding 'delta')
```

```
        we force it to update actual value to the target even if number of
steps is over.*/
        if (duty_cycle_actual != duty_cycle_new)
        {
            duty_cycle_actual = duty_cycle_new;
            pwm_update_duty_cycle(duty_cycle_actual);
            __LOG(LOG_SRC_APP, LOG_LEVEL_INFO, "Power updated to: %d.\n",
duty_cycle_actual);
        }
    }

}
```