Bluetooth Automation IO, and its place in a Cable Replacement Solution

Jakob Krantz
ijakkra@gmail.com

Kasper Bratz
dat12kbr@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Mats Cedervall (EIT), Mats Andersson (u-blox)

Examiner: Thomas Johansson

June 20, 2017

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Bluetooth Automation IO, and its place in a Cable Replacement Solution

With the rise of the Internet of Things (IoT) and a desire for connectivity between even the smallest devices, several low powered wireless technologies are fighting for a place in the IoT market. The Automation IO profile for Bluetooth Low Energy was designed to provide a low-level standardized way of exposing digital and analog inputs/outputs on a Bluetooth enabled device, and to give Bluetooth LE a chance in the IoT and automation market.

Automation IO is a relatively new profile, and there doesn't exist a proper evaluation of its uses or requirements. This thesis intends to bridge that gap by providing a thorough assessment of the Automation IO profile, its practical use cases, how to integrate it into an actual cable replacement module, as well as investigate its place in IoT and automation.

To examine whether Automation IO has a spot in IoT and automation, we first had to evaluate the underlying technologies. We did this by conducting a comparative investigation of Bluetooth LE compared to other low-powered wireless technologies. We also evaluated the Automation IO profile by investigating how to include it in an actual cable replacement module, how it interacts with an existing solution, as well as what practical use cases exist for the profile. By integrating an Automation IO Service into an existing cable replacement module, we investigate the requirements for such a module and provide hardware requirements and recommendations for how this integration can be accomplished.

The result of this thesis shows that Bluetooth LE is a powerful tool for connectivity in hardware restricted devices, and compares well with other low-powered wireless technologies. Bluetooth LE does, however, lack some of the benefits of mesh networking necessary for an extensive home automation system. The Automation IO profile is shown to provide a flexible, standardized protocol for exposing I/Os for most generic I/O modules. Our investigation also shows several benefits of having this as a standardized profile, rather than having vendor specific custom solutions.

Acknowledgements

Our thesis work was performed at u-blox and we want to extend our sincerest gratitude and appreciation to the employees for providing help and feedback throughout our thesis. In particular we want to extend our thanks to Mats Andersson, our mentor at u-blox, for providing valuable insight and direction as our thesis progressed.

We would also like to thank LTH and our supervisor at EIT, Mats Cedervall, for aiding us in planing and structuring our work.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ADC Analog-to-digital Converter
AIO Automation IO
AP Application profile
AT-Command Attention Command
ATT Attribute Protocol
BLE, Bluetooth LE Bluetooth Low Energy
BR Bluetooth Regular
ECDH Elliptic curve Diffie–Hellman
EVK Evaluation Kit
GAP Generic Access Profile
GATT Generic Attribute Profile
GPIO General Purpose I/O
I/O Input/Output
IOM I/O Module
IoT Internet of Things
MAC Media Access Control
MCU Micro Controller Unit
MITM Man-in-the-middle attack
MVP Minimal Viable Product
NFC Near Field Communication
OOB Out Of Band
PTS Profile Tuning Suite
REED Router Eligible end-device
SIG Special Interest Group
SPS Serial Port Service
SoC System-on-a-Chip
TC Trust Center
TI Texas Instruments
UART Universal Asynchronous Receiver/Transmitter
UUID Universally Unique Identifier
6LoWPAN IPv6 over Low-Power Wireless Personal Area Networks

Chapter 1
Introduction

Bluetooth Low Energy (Bluetooth LE, BLE) is a wireless technology intended to provide connectivity between devices where power consumption is restricted. With the recent inclusion of the Automation IO profile, adopted in July 2015 [1][2], as part of the Bluetooth core specification, it becomes interesting to investigate the usefulness of Automation IO and its place in automation and the Internet of Things (IoT).

1.1 Motivation

Bluetooth LE, adopted in the Bluetooth 4.0 core specification in 2010 [3], has allowed for Bluetooth connectivity in even the smallest-, battery powered-devices and gives Bluetooth a spot in the IoT [4].

The Bluetooth Special Interest Group (SIG) has defined several standardized profiles and services [5], each intended to solve a particular use case. Most of these profiles are, however, designed for a particular device, such as a thermometer or a fitness machine, and there has previously not existed a profile for low-level arbitrary communication. The purpose of the Automation IO profile is to attempt to bridge this gap in functionality by providing a standardized way for exposing low level digital and analog pins of a generic I/O module (IOM). In theory, the ability to monitor and control the pins of a generic IOM comes with endless possibilities and provides a low-level interface to almost any device. The only limit is what logic the hardware exposes over its pins that the cable replacement module is connected to. The primary purpose of this thesis is to evaluate what potential benefits can be found by having a standardized low-level interface to a generic IOM over Bluetooth LE and what benefits this might bring for IoT and automation purposes.

1.2 Scope

We performed this thesis at the short range division at u-blox, a company developing connectivity modules using Bluetooth, Wi-Fi, cellular, and positioning technologies to provide connectivity for a wide variety of applications. We were asked by u-blox to investigate how to integrate Automation IO into one of their cable replacement modules and what uses there could be for such a service in their modules. The primary concern stated by u-blox was that the solution had to be

able to run in parallel with the other services offered in the cable replacement module. Since the cable replacement modules have a tendency to be placed in hard to reach areas, u-blox also wanted us to investigate the best way to provision the Automation IO Service to a module. This work was also intended to produce a proof-of-concept solution to demonstrate the findings of our investigations.

The work conducted at u-blox is aimed to give a practical insight of how Automation IO can be used and integrated into existing cable replacement modules. In addition to this practical investigation, this thesis also includes a theoretical evaluation of the Automation IO profile as a whole. A part of this evaluation includes a comparative investigation of the competitive LE technologies, ZigBee and Thread. This theoretical part is intended to evaluate what place Automation IO has in the IoT and automation compared to other, more traditional, ways of exposing the pins of an IOM over a low powered wireless medium.

The theoretical investigation combined with the work done at u-blox should provide a good understanding of the Automation IO profile, how to use it in practice, and what place it will have in IoT and automation solutions.

1.3 Outline

The remainder of this thesis will be structured as follows:

- *Chapter 2: Theory* - Explains concepts that might be previously unknown to the reader and are required to understand the rest of this thesis.
- *Chapter 3: Methodology* - Details the scientific methodology of the work performed during this thesis and the reasoning behind our actions.
- *Chapter 4: Comparison of LE technologies* - A comparison of Bluetooth LE and other low energy wireless technologies.
- *Chapter 5: Automation IO, standardization or custom solution* - Evaluates the Automation IO standard and when it is suitable to use.
- *Chapter 6: Provisioning* - Explores different ways of provisioning low energy cable replacement modules.
- *Chapter 7: Prototype solution on a u-blox module* - Explains the practical work with building a prototype solution for an Automation IO Service in one of u-blox' cable replacement solutions.
- *Chapter 8: Results* - Present the results acquired during this thesis.
- *Chapter 9: Discussion and conclusions* - Covers our general thoughts on the outcome of this thesis, the work that was done at u-blox, and final thoughts on the Automation IO profile. The thesis is concluded by presenting our final thought on the result of the work on Automation IO, and whether it has a place in IoT, automation, and as a part in one of u-blox' cable replacement solutions.
- *Chapter 10: Future work* - Present what future work can be done with Automation IO, the prototype, and in general what future use cases there are for the profile.

Chapter 2
Theory

This chapter will present some useful theory and key concepts required to understand this thesis.

2.1 Bluetooth Low Energy

The information found in this section is primarily based on *"Getting started with Bluetooth low energy: tools and techniques for low-power networking"*[6].

The Bluetooth Low Energy wireless technology developed and maintained by the Bluetooth Special Interest Group is a standard intended to provide connectivity for devices at a considerably reduced power consumption compared to classic Bluetooth (BR, Bluetooth Regular). The Bluetooth LE technology is based on a duty cycle pattern, which is what allows the technology to be classified as a low powered technology. The idea behind the duty cycle pattern is to keep devices with a limited power supply in a low powered sleeping state whenever possible, only waking them up periodically to send or receive high intensity bursts of data. The duty cycle pattern is a pattern common to low energy devices, it allows a device to draw virtually zero power when idle, only spending power when there is data to send/receive. This behavior is ideal for devices that don't rely on a constant stream of data and powered by a limited power supply, such as a battery.

2.1.1 Bluetooth LE protocol stack

The Bluetooth LE stack is composed by two distinct detached parts [Fig. 2.1]: A controller and a host. The controller comprises the physical and Link Layer. The host comprises the upper layer functionality of the stack, such as the Link Controller and Adaptation Protocol (L2CAP), Attribute Protocol (ATT), Generic Attribute Protocol (GATT), Security Manager (SMP) and the Generic Access Profile (GAP). The host and the controller communicate through an interface called the Host Controller Interface (HCI) that acts as a two-way communication channel. User-defined applications and nonstandard profiles can be run on top of the host as shown in [Fig. 2.1].

The layers of the host can be summarized as follows:

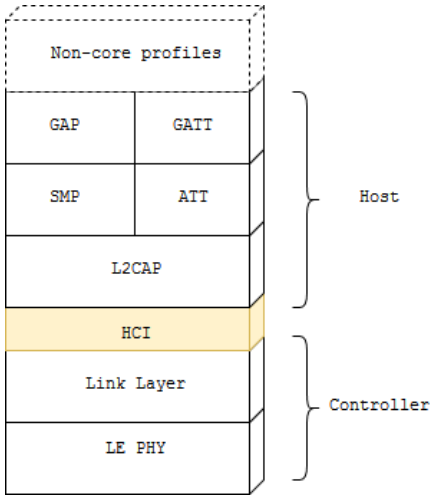

Figure 2.1: Bluetooth Low Energy protocol stack.

- **L2CAP** Transmits standard BLE packages to the controller. Also acts as the protocol multiplexer, responsible for routing incoming packages to either the ATT layer or SMP layer.
- **SMP** The security manager is responsible for the low level security operation needed in order to establish a secure connection, such as managing integrity during the pairing procedure, authentication, and encryption between Bluetooth LE devices.
- **ATT** Data in the top layers of the Bluetooth LE stack is based on attributes. An attribute is a data container coupled with identifiers and security parameters for the data. How these attributes are defined and how they can be accessed and interacted with is defined in the ATT layer of the stack. The ATT layer will be further discussed further in Section 2.1.2.
- **GATT** The top most data layer of the BLE stack. It encapsulates the ATT layer and adds structure and hierarchy to the attributes. This is often considered the backbone of BLE communication as it defines how data is organized and sent between different applications.
- **GAP** The top most control layer of the BLE stack. It defines things like BLE roles and how they interact with each other, security modes and levels, and control sequences; such as device discovery and connections.

2.1.2 Bluetooth LE roles

Bluetooth LE defines a couple of different roles depending on the abstraction level and scenario discussed [6, p.18, 36].

The Link Layer of the BLE stack defines four roles:

- **Advertiser** A device responsible for sending advertising packets, making it discoverable for scanners.
- **Scanner** A device scanning for advertisement packets.
- **Master** The device responsible for initiating a connection to a slave and maintaining it when connected.
- **Slave** The device that accepts connections from a master.

The GAP layer of the BLE stack also defines four roles. These roles always correspond to one or more of the Link Layer roles:

- **Broadcaster** A device optimized for regular broadcasting of data and corresponds to the Link Layer advertiser. A broadcaster device transmits data by sending advertisement packets, rather than requiring a connection.
- **Observer** A receive-only device responsible for scanning for packets sent by a broadcaster. An observer corresponds to the Link Layer scanner.
- **Central** A BLE central device corresponds to the Link Layer scanner and master. The Central is responsible for scanning for peripherals and initiating connections. The central is also responsible for maintaining several concurrent connected slaves.
- **Peripheral** A BLE peripheral corresponds to the Link Layer advertiser and slave. When a peripheral is not connected to a central, it transmits advertisement packets to help a central find and connect to it.

In BLE there is also the concept of Bluetooth servers and clients. These roles can not be tied to the GAP roles but are rather an indication of the current flow of information in the connection. In a BLE connection it is always the BLE client that requests data from a BLE server, and a BLE peripheral can depending on the use case act as either a client or server.

2.1.3 Network topology

A BLE slave can only be connected to one master while a master can manage up to seven active slaves. This is called a piconet and follows a star topology [7]. The size of a piconet is limited to eight active devices, but since a BLE slave want to spend most of its time in a deep sleep mode, a piconet can theoretically be much larger. In Bluetooth Regular a slave can be connected to several masters and in that way form a scatternet with unlimited size [Fig. 2.2]. This connection topology is currently not supported for BLE making it a so-called single hop technology [6].

2.1.4 Bluetooth ATT, GATT, and GAP

Bluetooth LE data is represented as attributes (ATT), each assigned a `16-bit handle`, a Universally Unique Identifier (UUID), a list of permissions and a value. The handle acts as a unique identifier for each attribute and works as an internal "addresses" for each attribute. The UUID acts as a type identifier for the attribute and specifies what kind of data it contains.

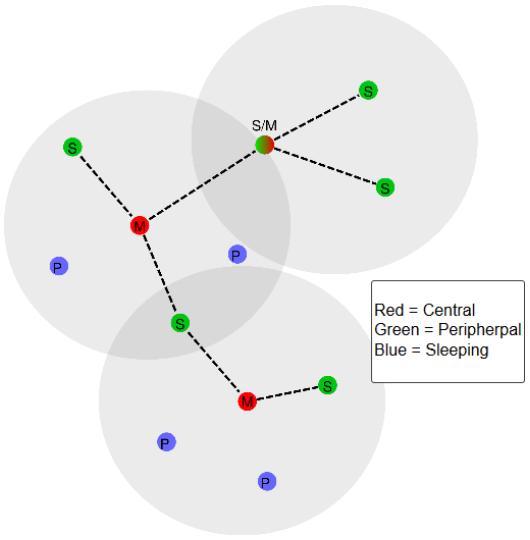

Figure 2.2: Illustration of a scatternet with three piconets.

	Handle	UUID	Permissions	Value
Attribute	0x0201	UUID (16 or 128 bit)	read/write/authentication	0x180A

Table 2.1: Example of a BLE attribute.

These attributes are grouped according to the GATT layer into services, characteristics, and descriptors [Fig. 2.3] where a server can contain many services, a service can contain one or more characteristics and a characteristic can contain zero or more descriptors. GATT attributes are stored in a table called a GATT-table, which is just a sequential table of the attributes contained in a GATT device. Attributes can not exist outside of this structure if the device wants to be compatible with other GATT devices, as this is the core of how data is represented in Bluetooth LE.

- **Service** A GATT Service is a group of related attributes, intended to represent the behavior of a part of a system. A Service is represented in GATT as a Service Declaration Attribute, followed by a list of characteristics and/or references to other services.
- **Characteristic** A characteristic acts as a data container for user data. A Characteristic Declaration (Containing metadata) and a Value Declaration (The actual user data) is used to represent a characteristic.
- **Descriptor** A descriptor is intended to provide additional metadata about a characteristic or the value associated with it. A descriptor is always placed within a characteristic and is comprised of a single attribute. For example, a descriptor can define the unit of the characteristic value.

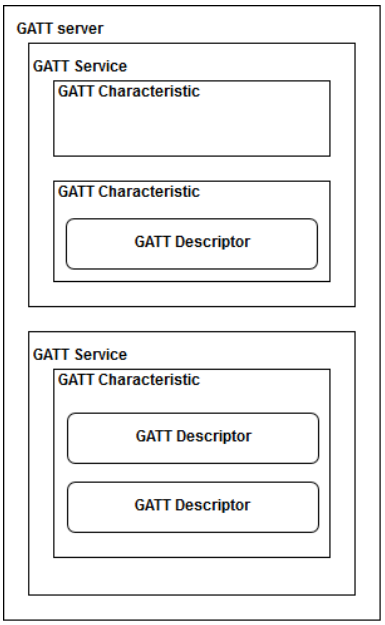

Figure 2.3: GATT server/service/characteristic.

	Handle	UUID	Permissions	Value
Service Declaration	0xAAAA	$\text{UUID}_{service}$	read	Service UUID

Table 2.2: Example of a declaration attribute for a service.

	Handle	UUID	Permissions	Value
Declaration	0xAAAA	$\text{UUID}_{characteristic}$	read	Properties, Value Handle Value UUID
Value	0xBBBB	Value UUID	Any	Actual value

Table 2.3: Example of a characteristic attribute.

Data flow

The GAP defines the topmost control layer and provides a framework that all Bluetooth LE devices must follow in order to discover each other, establish a secure connection and discover attributes from each other. Once two devices (A BLE client and BLE server) has connected and agreed on connection and security parameters, such as sleep times and encryption, a client can discover and read GATT attributes from the server. When a new connection is established, the client has no information about what data the server holds. In order to get data from the server, the client has to initiate a so-called discovery procedure to find what data the server holds. This allows the server to save energy by only transmitting data requested by the client. Communication between BLE devices is usually based on the client sending read/write requests to the server, allowing the server to stay in sleep mode as long as possible. Sometimes though, an asynchronous update from the server can be required. To get these updates without the client having to poll the server periodically (which cost both energy and bandwidth) there are two ways for the server to push packages to the client.

- **Characteristic Value Notification** The server sends a packet containing the value and handle of a characteristic without expecting any response. It is then up to the client to decide if it wants to act on the notification.
- **Characteristic Value Indication** Sends the same package as a notification, but requires an application level ACK message before it can send further indications. This approach is going to be slower, as only one indication can be sent each connection event, but will indicate if the message has been received or not.

2.1.5 Bluetooth Profile

Both Bluetooth Regular and Bluetooth LE defines something called a Bluetooth profile. Bluetooth profiles are often grouped with the Bluetooth protocol, but displays some key differences.

- **Protocol** A core part of all Bluetooth devices. It is the layers that include packet formatting, routing, encoding and other parts needed to send data between two devices.
- **Profile** Is either a part of the functionality that defines basic operations, such as reading, writing or representing data (GATT, GAP) or a way to achieve a specific use case. In essence, profiles define how protocols should be utilized to achieve a certain goal.

Bluetooth profiles are defined in such a way that a strict hierarchical structure can be implemented, and profiles can be built on top of each other. At the moment all Bluetooth LE profiles must implement both GATT and GAP as these are the top most data and control layers of Bluetooth LE communication, and no non-GATT profiles currently exist.

Bluetooth SIG defines a couple of use-case-specific Bluetooth profiles intended as a standardized way of accomplishing certain task [8]. Some examples include:

- **Find Me Profile** Allows the user to physically locate another device via BLE.
- **Blood Pressure Profile** Allows a BLE blood pressure reader to transmit readings wirelessly.

Users can also define their own profiles, keeping the standard for themselves or sharing it, allowing other vendors to provide implementations able to communicate with their products. An example of this is the Apple iBeacon where the profile is developed by Apple, but third party vendors are allowed to develop iBeacon compatible hardware transmitter based on the iBeacon profile.

2.2 Bluetooth Automation IO

Bluetooth Automation IO (AIO) is a Bluetooth SIG specified Bluetooth profile/service [1][2], adopted 14-July-2015 and is one of the core concepts of this thesis. The concept behind Bluetooth Automation IO is to expose digital and analog I/Os of a generic IOM and allow devices to monitor and interact with these I/Os. A GATT Server can be called an Automation IO server if it includes an Automation IO Service. An AIO client can then connect to an AIO server in order to interact with the service [Fig. 2.4].

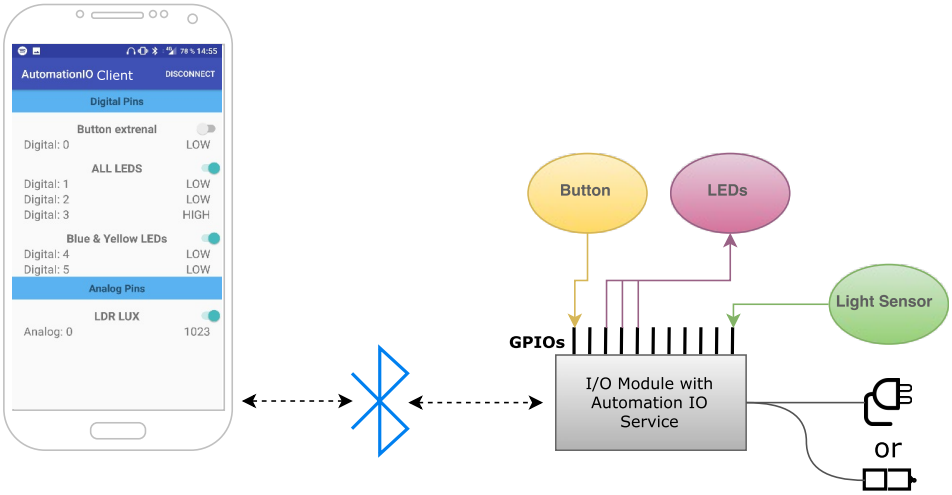

Figure 2.4: Illustration of an example usage of Automation IO.

The AIO Service, in turn, communicates with the IOM through the pins of the module and represents their values as characteristics. An IOM running the AIO Service work as seen in [Fig. 2.4]. The AIO Service is comprised of a collection of characteristics. An AIO characteristic can be one of the following types

1. **Digital Characteristic** A collection of one or more digital I/O signals can be grouped in a digital characteristic. The digital characteristic represents

the value of the signals with 2-bits/signal in the characteristic value field. A digital pin can only be high or low.

2. **Analog Characteristic** Is used to represent one analog I/O signal as a 16-bit value. The value is stored in the characteristic value field.

3. **Aggregate Characteristic** The aggregated characteristic is a characteristic that always represents the entire state of an IOM. This means the value of the aggregated characteristic is a collection of all digital and analog characteristics with the read property set. The aggregated characteristic is included as a way to read the entire state of an IOM with one read operation and in that way save both power and bandwidth.

Each AIO characteristic can have a certain number of properties describing how the characteristic can be interacted with, such as read/write permissions and if the characteristic can be indicated or notified. An AIO characteristic will also have a number of descriptors attached to it further describing the characteristic. These descriptors will contain information such as user defined descriptions, the number of pins in a characteristic and how the characteristic value should be presented.

Finally, a characteristic can have triggers attached to it. A trigger is a way for the AIO server to notify or indicate a characteristic when a certain trigger condition is met. There are two types of triggers

1. **Value Trigger** A trigger that triggers depending on the value of the characteristic. For instance, a value trigger can be set to trigger when an analog signal passes a certain threshold, or a certain number of pins have changed in a digital characteristic.

2. **Time Trigger** A trigger that triggers based on a time constraint, such as a time interval.

A characteristic can have either no triggers, only a value trigger, or one of each trigger.

The AIO specification is intended to provide a standardized profile for exposing generic IO signals. As any BLE client that implements the AIO profile can connect to any AIO server, AIO lets one device connect to many different devices, as long as they expose their data with Automation IO. This can be used in both an industry environment, to allow different industry robots to communicate when certain events transpire, or in a home automation environment to have one client read the state of all IoT devices in a household.

2.3 Thread

Thread is a low power, short range wireless device-to-device protocol, created by market leading companies in IoT, such as Nest, ARM, Silicon Labs and others [9]. Today Thread is managed by The Thread Group Alliance, which is a group consisting of members from the leading companies in IoT. Thread is built for IoT applications and the key selling points versus its competitors is the IPv6 based

communication, high security, robustness, mesh network capabilities and simplicity. Many of the alternative IoT solutions rely on one device acting as a communication hub, which routes all communication to other devices on the network. If this device goes down, the whole network stops working. Having a solution that avoids this behaviour was one of the main philosophy's when designing Thread [9].

The Thread stack can be seen in [Fig.2.5]. It is built on top of 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks). Thread adds UDP and IP Routing on top of 6LoWPAN. Thread, 6LoWPAN and many other low powered wireless technologies are built on top of `IEEE 802.15.4`. `IEEE 802.15.4` is a standard for low power and low data rate networks (other examples are `802.3` (Ethernet) and `820.11` (Wi-Fi)) and defines the physical (PHY) and Medium Access Control (MAC) layers of the OSI (Open Systems Interconnection) model. This allows Thread to be run on most hardware that supports `802.15.4`. In 6LoWPAN every device has it's own IPv6 address, allowing it to communicate with the Internet seamlessly. Most of the actual networking in Thread is defined by 6LoWPAN, such as mesh networking [9].

The Thread Network defines a couple of different types of devices. A Border Router is a router that act as a bridge from the Thread Network to adjacent networks, such as Wi-Fi or Ethernet. At the transport layer, the Border Router is transparent for devices outside of the network. A normal Thread Router provides routing between devices on the internal Thread Network. End-devices are devices running application code and act as the start/end points of the network. An end-device can either be a Router Eligible End-Device (REED) or a sleepy-end-device.

A REED is an end-device that can, if necessary, be upgraded to a Router. This can for instance be necessary if the REED is the only link connecting two part of the Thread network. A sleepy-end-device is a device that acts exclusively as a start/end point in the network and never as a router. A sleepy-end-device is connected to one parent node and all communication to and from the device is routed through that parent. A sleepy-end-device can sleep for long periods of time in order to save power. When the sleepy-end-device is sleeping, all messages intended for the device is stored in the parent router. When the device wakes up it receives those messages by polling the parent [9].

Inside of the Thread Network, header compression is used to reduce the overhead of IPv6. A normal IPv6 header is 40 bytes long. With header compression those 40 bytes can be reduced, how much depends on the scenario. The IPv6 header can be compressed to two bytes for a package send between two devices on the Thread Network. No matter the scenario, thread achieves at least a 50% header compression compared to traditional IPv6 headers that are 40 bytes [10, p.6-7].

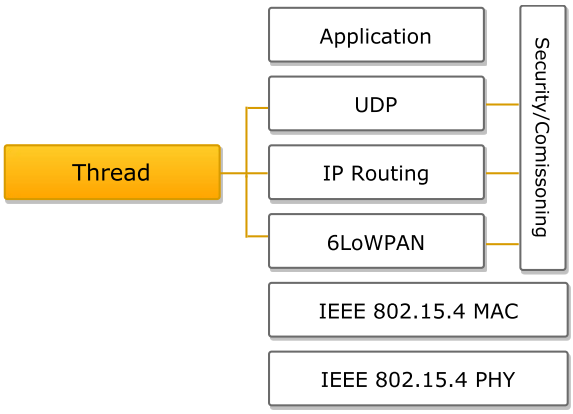

Figure 2.5: The Thread stack.

2.4 ZigBee

ZigBee is a standard for wireless, low-power, low-bandwidth networking, maintained by the ZigBee Alliance. There are many use cases for ZigBee, thanks to its mesh networking capabilities, the main ones are in IoT and home automation. ZigBee is, just like Thread, built on top of `802.15.4`, making it a good option for battery powered devices. A ZigBee end-device should be able to run on a small battery for an extended time.

The ZigBee Alliance is currently maintaining three official specifications of ZigBee: ZigBee PRO, ZigBee RF4CE, and ZigBee IP. ZigBee PRO is the default IoT networking standard for ZigBee devices and is used for device-to-device communication in large ZigBee networks. ZigBee PRO was originally just a feature set available for ZigBee 2007 with a collection of extra features, such as extended power saving, but is now the default standard for IoT networks. ZigBee RF4CE focuses on device-to-device communication without the fully featured mesh networking capabilities of ZigBee PRO. ZigBee IP uses IPv6 networking and adopts 6LoWPAN. When further talking about ZigBee and ZigBee networks, the ZigBee PRO specification is the one referred to.

A ZigBee network consists of a few different types of devices. A Coordinator is tasked with managing the network and there must exist one in every ZigBee network. The Coordinator is responsible for creating the network, choosing what channel to operate on and handing out addresses to new devices. It is also responsible for the security of the network, authenticating new devices and handing out network keys. A Coordinator can also act as a bridge to other networks. If the Coordinator would go offline, the entire network would also go offline and can be considered a single point of failure. The ZigBee network also consists of Routers, responsible for routing information through the network. Routers can also run application code and act similarly like the REED routers of the Thread network. The ZigBee network also has end-devices which are pretty much the same as sleepy-end-devices from the Thread network and follows a duty cycle pattern.

To send data to a device on the ZigBee network, the sender must have an

address to the receiver. Inside of the network, a two-byte address is enough for the Routers to know the destination, this address is handed out by the Coordinator when joining the network. It is also possible to address a device with a custom assigned Node Identifier e.g. a string; however, there is no guarantee of uniqueness. The last option is to use the unique four-byte serial number of the ZigBee device.

ZigBee, much like Bluetooth, defines profiles for certain use cases. ZigBee calls these profiles Application Profiles (AP) and act as a standard for interacting with the ZigBee stack. There are public and manufacturer specific APs, where public APs are specified by the ZigBee Alliance and manufacturer specific profiles are specified by the manufacturer [11].

2.5 AT-Commands

AT-Commands is part of the old Hayes Command set. Hayes Command set has been a long standing standard for issuing commands to modems (and other devices) over a serial connection and is today a general standard for issuing commands to embedded devices. The main reason AT-Commands is a common standard for smaller embedded devices is that it allows sending both arbitrary data as well as out of band control information over the same byte-stream channel. Sending data over a single line is accomplished by having devices listen for a special sequence of commands to indicate that it should switch to a "Command-mode" where it listens to further AT-Commands, rather than the arbitrary data sent otherwise.

AT-Commands are characterized by always starting with the letter combination AT, to indicate that the system should pay attention, and always end with a carriage return to indicate the end of the command. An AT-Command will return a status code, as well as an optional set of response parameters. Line feed and carriage return characters always enclose the status code and response parameters.

The remaining structure of an AT-Command would look something like:

```
[] = optional, <parameter_name> = parameter

AT+(command_name)
    [ = <parameter_name>,<parameter_name>,...] <CR>

[ <CR> <LF> Response,Response <LF> <CR> ]
<CR> <LF> OK/ERROR <LF> <CR>

// Real world example for setting Bluetooth
// discoverability mode of the NINA-B1 module
AT+UBTDM = <discoverability_mode> <CR>
<CR> <LF> OK <LF> <CR>
```

Chapter 3
Methodology

This chapter will explain the work process of this thesis, as well as the reasoning behind certain decisions. It will also discuss the problem description that laid the foundation for the thesis and how we intend to solve it.

3.1 Thesis goals

This thesis began as a thesis proposal from u-blox with a clear purpose, to investigate how to integrate Automation IO into one of their cable replacement modules. After further discussing the thesis with our mentor at u-blox the intentions of the thesis was further clarified, and we were able to state the following goals:

- Investigate how the new Bluetooth SIG specified Automation IO Service on top BLE can be used in u-blox products.
- Investigate how an Automation IO Service can operate as part of a u-blox cable replacement module. The investigation should examine how the Automation IO Service should interact with other services, how to enable and configure mapping to the I/O pins of the module, and how to interface with a module connected host MCU.
- Make a prototype implementation of the solution.

Our mentor at u-blox informed us that there had previously existed a custom solution for exposing I/O signals over Bluetooth Regular. The solution was never ported to their Bluetooth LE devices, but there still existed a need from customers for this type of solution. With the standardization of Automation IO, u-blox felt that it would be interesting to investigate if Automation IO could fill the role of their previous BR solution. These goals are, however, primarily focused on running Automation IO in u-blox products and to widen the scope of the thesis we decided on a few additional goals.

- Investigate whether Automation IO over BLE is a suitable option for exposing digital and analog I/Os of an IOM.
- Compare Automation IO to alternative implementations for other low energy wireless technologies.

- Investigate how using a pre-defined standard, rather than a custom solution, affect implementation complexity and usability of the final product.

We felt that these goals would provide a more thorough investigation of Automation IO and how to expose I/Os of an IOM over low power wireless technologies.

We started our thesis work at u-blox by studying the source code of the module we were going to work with, talking with other developers, and further study how evaluations of low powered technologies had been performed in the past. It quickly became apparent that we would have to modify our initial goals as some of them would not be feasible in practice.

Comparing how other technologies solved the same problem as Automation IO turned out to be hard due to a couple of factors. There was no standard for exposing I/Os for other technologies, and to fully evaluate Automation IO we would have to develop a custom solution for the other technologies as well. We also realized that the underlying logic would be the same for the other technologies and the only difference between the implementations would be how to send data over-air. Instead, we decided on a comparative investigation of different low powered technologies, and their place in automation and IoT.

We finally, together with our mentor at u-blox, agreed on a set of final questions that would be answered by the thesis, and a final set of goals that would have to be met to solve them.

- **Question 1.** How does Bluetooth Automation IO compare to other methods of exposing digital and analog signals, and what are the practical uses for it?
  - Conclude a comparative investigation of how Bluetooth LE compares to other low-powered wireless technologies and their place in automation and IoT.
  - Investigate how using the pre-defined Bluetooth AIO profile, rather than a custom solution, affects implementation complexity and usability when exposing I/Os.
  - Investigate how the new Bluetooth SIG specified Automation IO Service on top BLE can be used in a cable replacement module.
- **Question 2.** How can Automation IO be integrated into an existing cable replacement solution, and how can it interact with the rest of the system?
  - Investigate how an Automation IO Service can operate as part of a u-blox cable replacement module. Also, investigate how to interface the module to a host to provision the device.
  - Make a prototype implementation of the solution.

Answering these questions evaluates Automation IO from several different viewpoints and should accurately assess whether Automation IO has a part in automation/IoT applications.

We split the thesis in a practical part and a theoretical part, where Question 1 was primarily answered in the theoretical part, and Question 2 in the practical part.

3.2 Theoretical part

One of the issues we wanted to answer for this thesis was how Bluetooth Automation IO compared to similar solutions using other low powered wireless technologies. To answer this questions, we would use both theoretical knowledge of the technologies, as well as measurements on our prototype solution. We did change this goal into a general comparison between LE technologies instead. Due to time constraints, we would not have the time to implement similar solutions for the other technologies and the logic behind how to interact with pins and provision the devices would also be mostly similar. Instead, we focused on where the different solutions would differ the most, the functionality of the wireless technologies themselves.

We decided on two technologies other than BLE that we felt would give a good picture of how BLE compares to other low energy IoT and automation technologies.

- **ZigBee** A low powered wireless technology for mesh networking. We chose ZigBee for its widespread adoption in home automation devices.
- **Thread** A newly developed technology for low-powered wireless mesh networking over IPv6. We included Thread in our comparison as it is both a new technology, backed by several market leading companies, as well as its use of a communication protocol not often seen in low-powered technologies.

We decided to compare the different technologies mainly based on power consumption, memory requirement, and security. The reasoning behind these choices can be found in their respective sections in 4.

Another way to evaluate Automation IO is to evaluate the profile itself. This investigation will primarily focus on the general benefits and restrictions associated with introducing a standardized protocol in BLE and whether or not this is always suitable for Automation IO. We will also investigate how other technologies would handle exposing I/Os over-air and what standards are used by them. We will verify this goal with use cases, illustrating when the different approaches are suitable.

Originally one of the questions asked by u-blox was what uses Automation IO could have in one of their products. We instead generalized this question to an investigation of what use cases existed for Automation IO as a whole.

3.3 Practical part

For the practical part of the thesis, we created a prototype implementation of an Automation IO Service. The prototype was developed on a cable replacement module provided by u-blox. A detailed explanation of what the prototype will include and how we approached the implementation will be further discussed in Chapter 7. The prototype will be evaluated for correctness using the Bluetooth Profile Tuning Suite (PTS), a tool provided by the Bluetooth SIG for verifying profile implementations.

Integrating an Automation IO Service in an existing solution may sound like a trivial task, but there are several factors to consider before including new services.

- Automation IO can be run as the only service on a device, but in a real world setting e.g. a u-blox cable replacement module, the service is going to interact and be run in parallel with other services.
- The addition of Automation IO will place additional strain on power consumption, memory requirement, and throughput of the device.

We used our prototype to evaluate how our implementation of the service affected the module with regards to these concerns.

When building our prototype, we came across the question of how to properly initialize and configuring a device running Automation IO. The device running Automation IO is often a small embedded device that can be placed in a location where access is limited. For that reason, accessing or connecting to the module will be hard and a good provisioning routine is important. How to provision devices will be based on different approaches investigated when working with the u-blox module but could be generalized to the provisioning of other embedded wireless devices.

The prototype will act as one of our most valuable evaluation tools for the thesis and will help illustrate and verify other knowledge found during our work. The prototype will serve as a proof-of-concept for u-blox and help them evaluate whether to integrate an Automation IO Service in their cable replacement solutions.

Chapter 4
Comparison of LE technologies

To properly evaluate if Automation IO is a suitable candidate for cable replacement solutions or other IoT purposes, we first have to evaluate the technology behind it. In this chapter, we evaluate Bluetooth LE, as well as some of the other market leading, up-and-coming, LE technologies based on their IoT and automation capabilities.

4.1 Battery consumption

Bluetooth LE, ZigBee, and Thread can all be gathered under the common name: Low Powered Wireless Technologies (LPWT). As the name implies, one of the most important characteristics of these technologies is that they are energy efficient. LPWTs are often used in portable devices, such as a mobile phone, or in IoT applications, such as sensors or controllers. In both of these cases, it is common that the device is run on battery, as a power connector is not a feasible solution. It is also desirable that the device can be sustained an extensive amount of time on one battery, as having to replace batteries too often can be both tedious and expensive.

All of the technologies investigated in this thesis are classified as LPWTs, but the question is how energy efficient they are compared to each other. The difference between 1 or 4 years expected battery life might not sound like it would matter, but it can be the difference between changing 5 batteries/month or 20 batteries/month. As previously mentioned, these devices can be in hard to reach places and replacing a battery can be a difficult task.

When we investigated power consumption for BLE, ZigBee, and Thread, as well as every other low powered technology we came across, it quickly became apparent that all of them utilized a duty cycle pattern. A duty cycle pattern is when a low powered device is kept in an ultra-low-power mode majority of the time, only waking up to send/receive messages. How the duty cycle pattern, as well as how the technology handles connecting/scanning for other devices, appears to be the key factors when determining the energy consumption of a LWPT and is what we investigated in this section.

To understand how battery consumption is handled, a few keywords have to be defined:

- **Connection Event** When a slave and master both wake up and exchange data is called a connection event.
- **Connection interval** The time spent in sleep mode between the start of two consecutive connection events.
- **Slave latency** The connection interval specifies the time between two connection events and is agreed upon by both connected devices. Slave latency allows the end-device/peripheral, which is often the weaker device, to skip connection events if it has no data to send, but enables it to push data faster if need be.
- **Rx/Tx mode** The state when a device is able to Receive (Rx) or Transmit (Tx) data.

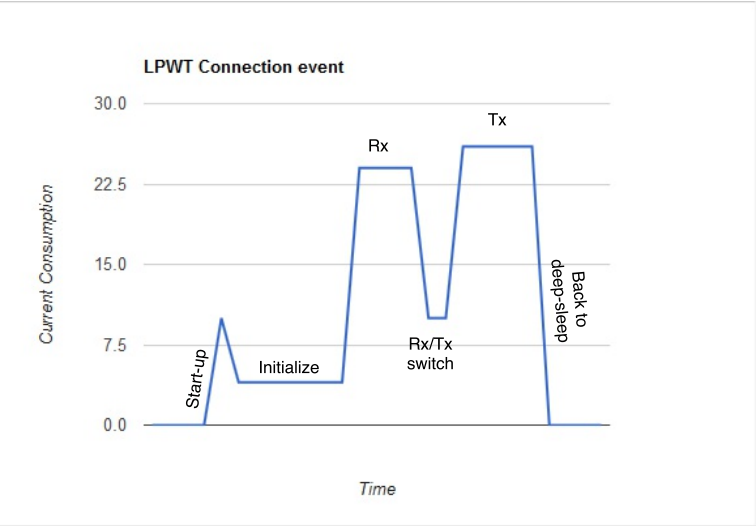

Figure 4.1: A typical connection event for a LPWT.

The way Bluetooth LE handles connectivity is to let a master device scan for possible connectable slaves, while the slave periodically sends out what's called advertisement packets. This behavior allows the peripheral to keep its duty cycle pattern even when advertising, only advertising its availability when suitable, leaving the expensive scanning to the central device. When a connection is established the two devices agree on connection parameters, such as connection interval and slave latency, then settles into the duty cycle pattern. The connection interval is decided between both devices, meaning the central will always attempt to establish a connection event every connection interval. If transmission speed is not crucial, the peripheral can skip a certain number of consecutive connection events, specified by the slave latency parameter, if there is no data to transmit [6].

A typical connection event [Fig. 4.1] for a device consists of a wake-up and initialization phase, where the device starts back up and re-connects to a paired device. The device then transitions into an Rx and Tx event, where the devices

convey whether they have data to transmit or not. The devices will then alternate between Rx and Tx mode, communicating relevant data to each other. The devices will then go into a post-processing phase eventually going back to deep-sleep [12, p.26]. The duty cycle, or the average wake time of a device during a connection interval, as well as how much energy is used in each of these phases, will be what determines the power consumption of a connection interval.

ZigBee handles connection events almost the same way as BLE [13]. The main difference in power consumption between the two technologies will come from the duration and power consumption of the different phases of the connection event. When ZigBee devices are scanning for connections, it is the ZigBee end-device that is responsible for scanning and initiating a connection, rather than the routers. This behavior is more energy demanding as a scan event is more costly than an advertising event [14].

Thread end-devices also use the duty cycle pattern. Sending a message to a Thread end-device will get routed through the Thread network, finally arriving at the router connected to the destined end-device. The message will then be stored in the router for a set amount of time, before being discarded or retrieved. When a Thread device wakes up, it polls its connected router for messages, and if there are messages queued up they are sent in alternating Rx/Tx modes. If the Thread device wants to send a message to its router, it simply forms a thread package and wakes up to send it [15]. Since Thread routers are usually not powered by a battery, there is no need to maintain a shared connection interval value in a Thread network and the end-device decides for itself how often it should wake up.

Another factor when talking about power consumption is how the technologies handles interference. Packages in transfer that are lost or damaged have to be resent, increasing the time of connection events. Bluetooth manages interference by using frequency hopping, a technique used to switch between frequency channels to avoid collision and interference. ZigBee and Thread are both built on top of 802.15.4 and thus uses CSMA/CA to handle interference. CSMA/CA is a way for multiple devices to transmit data on a single channel while avoiding collisions. Using CSMA/CA will slightly prolong the time ZigBee and Thread devices has to maintain a connection event [14].

4.2 Memory requirements

BLE, ZigBee and Thread are often run on embedded systems with limited hardware. One restriction when working with limited hardware is the limited amount of flash and RAM memory and a wireless technology with high memory requirements will not always be a valid option. Choosing a stronger MCU for a device just because it has more memory would not only be excessive but could also become costly. This might not appear crucial for smaller projects, but when manufacturing thousands of units, companies can save a lot of money by choosing the correct MCU. By choosing a wireless technology which require less memory, the device might be able to be run on a cheaper MCU.

Wireless technology stacks don't have a definite memory requirement, as they are developed by different manufacturers and memory requirement will depend on

how the stack is implemented. Most of the stacks can be configured at compile time to include/exclude certain features. Since both size and compilation options vary between manufacturers it was hard to directly compare the different technologies.

For our comparison we have summarized the memory requirements for the different stacks, with a couple of different compilation options. We found data for the ZigBee stack from Texas Instruments (TI), the Bluetooth stack from Nordic Semiconductor, and for the Thread stack we ourselves compiled and analyzed OpenThread, the first Thread Group certified Thread stack. When analyzing the Thread stack we studied the .elf files using the **arm-eabi-size** command from the GNU ARM Embedded Tool-chain and was able to get the required flash and RAM memory requirements. The memory requirements will also vary depending on the compilation target, so the targeted chip-set is also included in our measurements.

The memory usage for the different stacks can be found in Tables [4.1, 4.2, 4.3] [16][17][18]. The comparison contains two different versions of Nordic Semiconductor's stack, S110 which only supports peripheral and S132 which support all 4 Bluetooth LE roles, up to eight connections and concurrent role operations. Data for the ZigBee stack comes from TI and includes a configuration for whether to include the PRO functionality set and/or the secure option. The PRO functionality includes several newer enhancements compared to the regular ZigBee stack and will need more memory. The secure option includes whether to include authentication, encryption and other security options. The memory requirements measured for OpenThread is based on the stack required for a REED router. The required resources can be reduced significantly for a sleepy-end-device, which is not router-eligible. The decision to study the memory requirements for the REED and not every separate device, is purely based on convenience/time.

ZigBee and Thread have statistics for both with and without security, the Nordic Semiconductor BLE stacks doesn't have that option. Not all data are comparable between the different stacks and as mentioned earlier it is hard to give a fair comparison when there are so many factors included. There are, however, some conclusions that can be drawn, which will be further discussed in Section 8.1

		Coordinator	**Router**	**End-Device**
PRO	Secure	Flash/RAM	Flash/RAM	Flash/RAM
on	on	158.4K/6.7K	158.2K/6.7K	128.2K/5.3K
on	off	149.2K/6.7K	148.0K/6.7K	120.1K/5.2K
off	on	147.1K/6.6K	146.9K/6.6K	121.9K/5.2K
off	off	137.8K/6.6K	136.6K/6.6K	113.8K/5.2K

Table 4.1: Memory usage for TI's ZigBee stack (Z-Stack) compiled for Texas Instrument CC2530 module.

SoftDevice/SoC family	Functionality	Flash	RAM
S110 8.0/nRF{51822/51422}	Single mode peripheral.	92KB	8KB
S132 3.0/nRF52	Every BLE role. 20 connections	120KB	8KB

Table 4.2: Memory usage for Nordic Semiconductor's BLE stack.

Security	Flash	RAM
off	51KB	14KB
on	68KB	15KB

Table 4.3: Memory usage for OpenThread REED, developed by NEST, compiled for ARM Cortex-M3 target.

4.3 Security

In a world where we are more connected than ever, and even your toothbrush can be connected to your phone, it is highly important that even the smallest devices are securely connected. It becomes even more important when you consider an unknown entity could compromise or manipulate connected devices such as health monitors or even your car.

For low powered connectivity devices such as BLE, ZigBee and Thread, there are going to exist a couple of main attack surfaces.

- **Passive eavesdropping** The process of a third party device listening to the data being sent between two devices.
- **Man In The Middle Attack (MITM)** A third party device impersonates the two other devices in order to get the other devices to connect to it instead. The third party can then route the communication between the two devices. This will give the illusion that the two original devices are actually connected to each other while allowing the third party device to block or inject messages between the devices.
- **Identity tracking** BLE also have the problem of Identity tracking [19] where a malicious entity associates a BLE address with a person and physically track that person based upon the presence of the BLE device. This is less of a problem for ZigBee and Thread, as these devices don't tend to be carried around by people.

Bluetooth manages the problem of identity tracking by periodically changing the address of a BLE device. This stops a malicious entity from following a BLE device solely based on the proximity of a device with a certain address.

Passive eavesdropping is a problem that can be mitigated by encrypting data in transfer. All the technologies we investigated encrypts their data with 128-bit AES cryptography, an encryption standard classified by the National Institute of Standards and Technology (NIST) as secure for federal government applications [20, p.37].

For MITM attacks a strong encryption might not be enough, as the encryption will be rendered useless if a malicious entity can intercept the key exchange process. How key exchange is handled when joining a new network is crucial to prevent MITM attacks and is also where the different technologies differ the most.

- **BLE** Before a BLE device can communicate with another device, a pairing procedure is carried out. It is in the pairing procedure the device exchange device information in order to establish a secure connection [19]. The security of the pairing procedure is determined by what's called security modes and levels. BLE supports two different security modes
  - `Security Mode 1` Enforces security by means of encryption. It has four different levels [6, p.45][19].
    - **Level 1** The link is not encrypted.
    - **Level 2** The link is encrypted with AES-128, the key exchange is not authenticated.
    - **Level 3** The link is encrypted and the key exchange is authenticated with Secure Simple Pairing methods.
    - **Level 4** The link is encrypted and the key exchange is authenticated with LE Secure Connections pairing.
  - `Security Mode 2` Enforces security by means of data signing. It has two different levels [6].
    - **Level 1** Unauthenticated data signing.
    - **Level 2** Authenticated data signing.

  For security mode 1, obviously, level 1 offers no security and the connection is vulnerable to both passive eavesdropping and MITM attacks. Level 2 offers more protection against passive eavesdropping as long as the third party has not acquired the decryption keys. Security Level 3 and 4 both require the key exchange to be authenticated by both devices. The difference between level 3 and 4 is that level 3 have to be Bluetooth legacy compatible, using a custom key exchange protocol, while level 4 uses Elliptic Curve Diffie-Hellman (ECDH) key exchange. The authentication methods provided by BLE are
  - **Pass key** An identical six-digit passkey is displayed on both devices and have to be verified before the key exchange is verified.
  - **Out Of Band (OOB)** Another wireless technology is used to exchange keys, such as Near Field Communication (NFC). The security of this method is entirely dependent on the OOB technology security.

  Level 4 also provides an authentication method called `Numeric comparison` based both devices generating confirmation values based on nonce data from the other device. This value then has to be confirmed by a human before the connection is approved. Numeric comparison is considered the most secure authentication method for BLE. Depending on what authentication method is used, level 3 and 4 offers high resilience to both MITM and passive eavesdropping attacks.

Security mode 2 does not really protect the key exchange process as it is more directed towards data integrity than data confidentiality and does not really stop any of the attacks. Mixing security mode 1 and 2 should give a highly secure connection.

- **Thread** Security in Thread is based on TLS communication using AES. The TLS secret is agreed on by using `Elliptic Curve J-PAKE`. J-PAKE is a password-authenticated key exchange protocol with "juggling". What this means is that Elliptic Curve Diffie-Hellman is used for the key agreement process and Schnorr signatures as proof to authenticate the agreement [21]. Thread also uses a network-wide key in order to prevent eavesdropping and targeted disruption.

  The process of joining a Thread network is called a joining process. The components in the joining process are going to be a connecting device, a Border Router, a Joining Router and a Commissioner. The Commissioner will be a device controlled by an administrator of the Thread network and is the device that authenticates the joining device. The Joining Router is responsible for, if the new device is approved by the Commissioner, handing out network keys and network addresses to new device. A very basic join process can look something like this.

  1. The connecting device sends a connection request to a Border Router.
  2. The Border Router, without allowing the connecting device into the network, forwards the request to a Joining Router.
  3. The Joining Router sends a request to a commissioning device. This device can be something like a phone or Thread controller in the users possessions.
  4. The user physically verifies that some alphanumerical passkey provided by the connecting device is displayed on the commissioning device. This passkey is something that can also be verified physically on the device, such as a serial number.
  5. If the user verifies the passkey the connecting device is allowed by the Joining Router to join the network.

  These steps can sometimes be trivialized as one device in the network can assume several roles in the process. This approach is highly resilient to both passive eavesdropping due to the use of AES and TLS, and MITM attacks due to its key exchange and joining protocols.

- **ZigBee** Security in a ZigBee network is based on a three-layer security model [22] working on the MAC, Network (NWK) and Application (APL) layers. In a ZigBee network, there will always be one, and only one, device known as a Trust Center (TC). The TC is responsible for distributing and managing keys in the network, as well as authenticating untrusted join requests. In a ZigBee network, there exist two types of keys.

  – **Link-layer Key** Is used to secure unicast communication on the network [23] and is a key directly established between two devices.

- **Network Key** A network-wide key used for broadcasting information on the network. This key is always shared by all devices in the network.

ZigBee devices always come pre-installed with default symmetrical network and Link Layer keys so that unencrypted keys never have to be sent through the network. This, however, is entirely dependent on the safekeeping and handling of these pre-installed keys. If for instance the network key would be compromised through one weak device in the network, the entire network would be compromised [23]. ZigBee has one exception to the rule about never sending unencrypted keys on the network. This weakness is presented when a non-preconfigured device joins the network, as one unencrypted key can be sent to allow secure communication to this device. This one-time exception leaves a small time frame where a malicious entity could sniff the key and compromise the network [23].

ZigBee can be considered a safe network, resilient to both passive eavesdropping and MITM attacks, as long as keys are properly managed and stored. However, using pre-installed keys and sending unencrypted keys on the network present some potentially critical security risks.

Chapter 5

Automation IO, standardization or custom solution

Transmitting data over a wireless medium can be done in various ways. In BLE, profiles are used to define how different use cases or scenarios can be solved using the BLE stack. With the adoption of the Automation IO profile into the Bluetooth standard and the apparent interest shown by u-blox, one begins to wonder what's the advantage of having a standardized profile and what keeps companies from defining custom profiles if there is a demand for the functionality. The Automation IO profile and the possibility of controlling I/O signals wirelessly could open up new opportunities in IoT, home automation, and in the industry. Providing a standardized way to control and monitor the signals of a generic IOM, as long as a BLE module can connect to it, could save both time and money for the end-users.

This chapter will investigate the impact of using the standardized Automation IO profile for exposing I/Os compared to a custom BLE profile and when each solution is most suitable.

The Automation IO profile is designed in a way that any IOM should be able to be represented, and this brings with it both benefits and caveats. To fulfill the requirement of being able to represent any IOM, the profile has to be rather comprehensive and offer many options for representing a signal. One of the strengths of the Automation IO profile is its flexibility. Many of the requirements of the profile are marked as optional [1][2] and there is no limit for how many characteristics there can be in the service, except for the size of the GATT-table in the BLE stack. The end-user can themselves chose what subset of features they want to support in their implementation of the profile, allowing the final product to be anything from very barebones, to very comprehensive and adaptable.

Automation IO's claim of being able to represent any generic IOM, while true, might not always the best option. Automation IO is a predefined profile, and the functionality offered by the profile may or may not fulfill the needs of a vendor. As Automation IO is a predefined profile, it prohibits users from adding or removing functionality, such as adding additional descriptors, if they want to claim compliance with the profile. Perhaps only a small subset of features of Automation IO is required, and a barebone implementation is still too excessive, but exposing I/Os through Bluetooth is still the required behavior. Automation IO could also provide too little functionality, such as if the user wants to process a signal before transmitting it. Automation IO only supports exposing the current value of a pin

and thus processing the signal is not allowed.

Fortunately, it is possible to define custom services in Bluetooth. Depending on the use case, a custom service may be easier to implement and introduce less complexity in the system. A custom service will also allow the implementation to be better customized to the needs of an individual device. A custom service will, however, require the definition of a custom profile, which will require knowledge and time, and could in the end cost a company more money. Deciding on using a custom solution, not following the Automation IO standard, will also prevent the service from claiming to be an Automation IO Service, which comes with its own limitations.

The benefit of following a predefined profile is that it acts as a contract for what the profile can and can not do. If a service claims to follow a predefined standard, it will always contain a certain set of characteristics and descriptors, comply with certain security requirements, and behave in a deterministic way. Predefined services allow the development of generic profile-based peripherals and centrals. One example of this would be that an AIO central always know an AIO peripheral will contain nothing other than digital, analog and aggregated characteristics. That behavior might allow a vendor to use a generic client, such as a mobile application, to monitor their device running their version of an Automation IO Service. This is true in general for any Bluetooth profile, as using a standardized profile comes with the benefit of knowing that other devices supporting the same profile will be able to interact with each other.

For comparison's sake, we also investigate how ZigBee and Thread handle standardizing use cases. ZigBee APs can be directly compared to Bluetooth profiles. ZigBee does however not offer an AP for exposing I/Os and building a custom profile would be necessary. Thread does not provide an application layer and designing a custom protocol for how to represent pins would be needed. Not having an application layer would, however, allow for a super barebones solution where a simple 1/0 could be used to control an output pin if that was the sole purpose of the Thread end-device. ZigBee Alliance and The Thread Group have worked together to get the ZigBee application layer running on top of Thread [24]. Thread could theoretically benefit from ZigBee APs as well, and if an Automation IO profile becomes available for ZigBee, it would also become available for Thread.

What approach is "the best" when exposing data over a wireless medium is going to be entirely dependent on use case. A standardized Bluetooth profile brings with it many benefits but also forces the user to comply with the specifications of the profile. In Chapter 8.2 we will analyze some use cases to see whether a standardized or custom profile is the best option.

Chapter 6
Provisioning

An interesting question we came across when framing the issues for this rapport was how to best provision a device. Provisioning is the act of preparing or equipping a network to allow it to provide a new service. In our case, this was the act of how to initialize an AIO server to represent a generic I/O module. This might seem to be a trivial problem, but when dealing with IoT devices where the BLE module is placed in a hard to reach places with limited options for connecting, provisioning becomes an important question.

Some interesting metrics to consider when talking about provisioning are

- **Accessibility** How easy is it to connect to the module in order to push new settings?
- **Simplicity** How does the user push settings to the module? Does the user have to know how to code or is it a simpler interface?
- **Adaptability** How easy is it to modify settings when the module is running?
- **Memory** Since the device might run on limited power it is also interesting how the service handles powering off.

We considered a couple of different approaches to provision an AIO server and what benefits and restrictions come with them.

- **Hard coded** One way to provision a module is to embed the settings in the firmware of the BLE module. This approach is appropriate if the layout of the IOM is previously known, as well as what characteristic and settings will be required. This approach will store the settings in the flash memory of the device and will persist when the device is powered off. This approach, however, requires the device to be re-flashes if the IOM or the desired structure of the service is changed. These changes would also have to be made directly in the firmware source code, which is usually some low-level programming language.
- **Setup script** Setting up the Automation IO Service with a user defined script, run at device power on, could be one way to provision a device. This would, unlike having the settings for the server hard-coded in the firmware, allow the user to define the service in a higher level scripting language. This approach could also be modified to let the device run a configuration

document rather than running a scripting engine. This would limit the logic applied to the service, but would save memory that would be required to run a scripting engine. This approach would abstract and simplify the configuration of the device, but would still need to be re-uploaded in order to push changes.

- **Transmitting data via UART** If the BLE module has a Universal Asynchronous Receiver/Transmitter (UART), serial data could be pushed from the connected host (if there is one) to set up the Automation IO Service. This would allow the user to customize the initialization of the module to the exact needs of the specific module. This approach would allow for a custom command set to be used when setting up the service, such as Attention-Commands (AT-Commands), which would be more user-friendly and not require the user to know low-level programming. These commands would also be sent asynchronously which would allow the service to be modified even while running. This approach does obviously rely on the module having a UART connection to a module connected host. Unless specified by the firmware, this approach will not provide persistent settings and will have to be re-provisioned on power on.
- **Transmitting data wirelessly** Sending data, such as AT-Commands, wirelessly would display many of the same benefits and restrictions as for sending data over UART. The main difference is that sending data wirelessly would eliminate the need for a UART connection and even a module connected host. Instead, the module would have to include a service dedicated to receiving commands over BLE, such as the u-blox Serial Port Service (SPS) [25].

There will, of course, be more ways to provision a device, but these are the ones we investigated during this thesis. The best way to provision a device is going to be dependant on the situation, as we will see in Chapter 8.2 where we are going to evaluate some common use cases and analyze what choice of provisioning is best for each case.

Chapter 7

Prototype solution on a u-blox module

We conducted our thesis work at u-blox, a company working with wireless semiconductors and modules in the automotive, industrial and consumer markets. Their technology is based on wireless communication and positioning using short range radio (Wi-Fi/Bluetooth), GPS or cellular networks. Our thesis work was based in the Malmö office which is primarily working on developing short-range Wi-Fi and Bluetooth chips/modules. Part of our thesis work involved integrating an Automation IO Service in one of u-blox' cable replacement modules.

We decided, together with our mentor at u-blox, that the NINA-B1 module would be suitable for this work. The NINA-B1 module is a standalone short range Bluetooth low energy module that runs Bluetooth V4.2 and comes pre-flashed with u-blox' connectivity software. The module, however, offers full support for running customer developed applications directly on top of the BLE stack.

When developing our service we used a NINA-B1 Evaluation Kit (NINA-B1-EVK) which is an evaluation board mounted with a NINA-B1 module. The EVK gave us a simple interface to the module, allowing us to access the module pins as well as giving us a simple serial interface letting us debug and re-flash the module from a computer [Fig. 7.1]. Our initial plan was to develop a standalone application running an Automation IO Service on the module. However, since one of the goals of this thesis was to evaluate how Automation IO could be integrated into an already existing solution, we instead decided to build our prototype solution on-top of u-blox' existing connectivity software.

7.1 System overview

The u-blox connectivity software is the default firmware for the NINA-B1 and what we used as a base when integrating Automation IO in the NINA-B1 module. The connectivity software is an extensive piece of software with the goal of providing users a high-level interface to the NINA-B1 module where they can connect the module to other BLE devices acting as either a central or peripheral, customizing it the way they want to use it. AT-Commands (described in Section 7.3.2) are used to interface with the module and allow the user to perform device discovery and connections, but also to set up, read, or write to any standard or custom BLE service.

AT-Commands can either be sent as serial data through the UART, from a

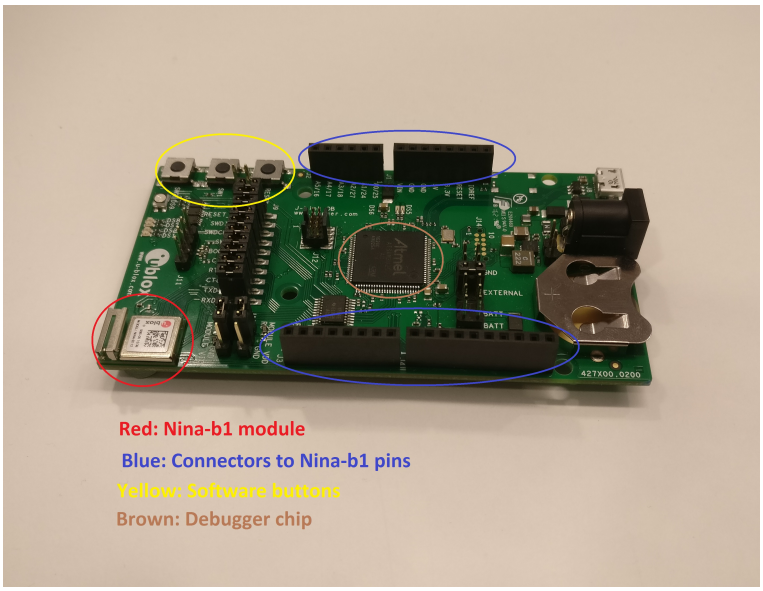

Figure 7.1: A NINA-B1 evaluation kit.

module connected host or wirelessly with the SPS. The SPS is one of u-blox' custom BLE service designed to send and receive serial data streams over BLE.

AT-Commands can also be used to configure low-level settings for the module, such as connection intervals, advertising settings and security modes and levels.

[Fig. 7.2] show an overview of a typical system, using u-blox' connectivity software.

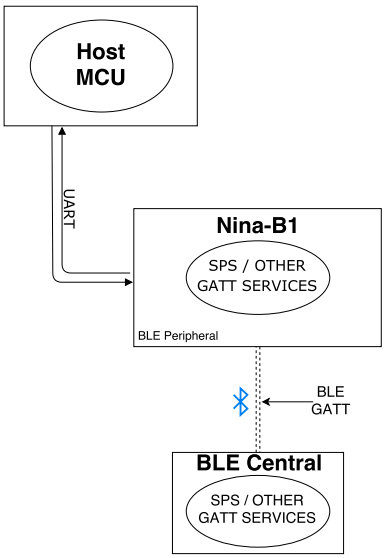

Figure 7.2: An overview of a system using the connectivity software.

Integrating Automation IO in the current version of the connectivity software rather than building a standalone application came with a lot of benefits, but also some restrictions and limitations we had to work around. The connectivity software, as previously mentioned, comes pre-built with functionality for device discovery, connecting to other devices, etc. allowing us to fully focus on the core of this thesis, Automation IO. The downsides, however, of working on such a vast and complex system is that it is not necessarily built to support the functionality you want to integrate and additions had to be made in some parts of the system. When discussing how to integrate Automation IO with the rest of the system, it was requested that the service would work in parallel with the SPS and other GATT services, allowing the Nina to run several services at once, without disturbing each other [Fig. 7.3].

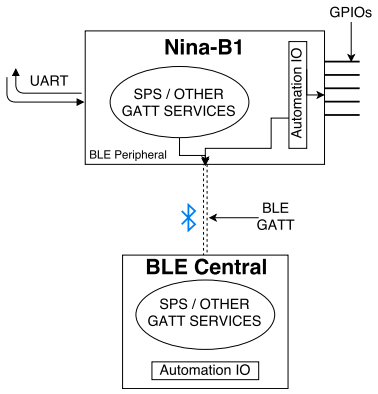

Figure 7.3: An overview of how the module should work with Automation IO integrated.

7.2 The process

Together with our mentor at u-blox, we decided on some goals for the implementation process.

1. Our first goal was to implement a Minimal Viable Product (MVP), a barebones implementation of the Automation IO Service, with a hard coded static characteristic representing the value of a single pin. The MVP was intended to give us a good overview of the system, as well as an estimation of the complexity and time requirements of working on the NINA-B1 module.

2. The second step in the process was to provide full support for digital characteristics, as they appeared easier to implement than analog characteristics. This activity included allowing users, via AT-Commands, to define characteristics and what pins they would represent, as well as having a characteristic representing a group of digital pins rather than just a single pin.

3. In the third step of our implementation, we include both analog and aggregated characteristics. With the inclusion of these characteristics, the AIO Service could now represent any arbitrary IOM.

4. The fourth and final step would be to implement time and value triggers. As there are four time-trigger settings and eight value-trigger settings and they each interact with each other differently, we decided on some key triggers to include and focused on them. By this point, the service would fully comply with the Automation IO Service and profile specifications [1][2], but not necessarily implement every optional requirement.

5. We also developed an Android application, able to connect to and control an Automation IO Service. The application would give us a way to test our system, as well as act a tool to illustrate the use of an Automation IO Service. We developed and updated the application in parallel with the Automation IO Service in step 3 and 4.

We set these goals as the baseline for our project and the final aim of the prototyping stage of the thesis. We felt that by completing these milestones, we would have a product that could properly showcase the benefits and restrictions of AIO and how to integrate it into an existing cable replacement solution.

7.3 Implementation

This section will go through the steps and design consideration we took when implementing the Automation IO Service on the NINA-B1 module. We will also discuss the problems we faced when implementing the prototype, as well as how we handled them.

7.3.1 Implementation tools

U-blox uses Visual Studio and Visual GDB to develop and flash firmware onto their modules. Visual GDB is a plugin for Visual Studio that allows us to flash and debug firmware, using the onboard J-Link hardware on the EVK. NINA-B1 can be programmed both with ARM mbed and nRF5 SDK. We used nRF5 SDK for our prototype since that is the SDK used for the rest of the connectivity software. The SDK is developed by Nordic Semiconductor; a company focused primarily on the development of ultra-low-powered wireless Systems on Chip (SoC) solutions. The nRF5 SDK is a software development environment for their SoC. It contains everything needed to interact with the hardware and the BLE stack.

To configure and test our module we used a program called s-center. The s-center software is developed by u-blox and is a tool to configure, evaluate and test their short-range modules. It is a graphical application interface that we used primarily to send AT-Commands to our module.

7.3.2 AT-Commands

As s-center and the connectivity software of the NINA-B1 module already featured full support for AT-Commands, we naturally decided to use AT-Commands to provision the Automation IO Service as well. The connectivity software featured an AT-parser, with rules for what commands should be accepted. It was a simple process to extend the rule-set for the NINA-B1 to allow the commands we felt was necessary to provision our service and pass the parameters on to whatever function we decided.

We defined four custom AT-Commands for the Automation IO Service. Three commands to add characteristics, one for each type of characteristic, and one command to add triggers to a characteristic. It was necessary to provide one AT-Command for each type of characteristic as the parameters for each type were different. Each of these commands allows the user to configure what security permissions should be set for the characteristic, whether to use indications or notifications and what pins should be bound to each characteristic.

7.3.3 Automation IO server

Automation IO is a fairly big profile, with many mandatory and optional requirements. Those requirements result in a fair amount of necessary logic. The final-, high-level- design for the implementation of the Automation IO Service is illustrated in [Figure. 7.4]. In the initial design for the MVP, the structure was considerably different, as we placed all of our logic in the `AIOService`. The purpose of the MVP was to get to know the SDK and existing software. It also helped us to estimate how long the service would take to implement. After progressing past the MVP, the code base quickly grew and as we added more functionality, like triggers and dynamic characteristics, we had to expand our design of the system.

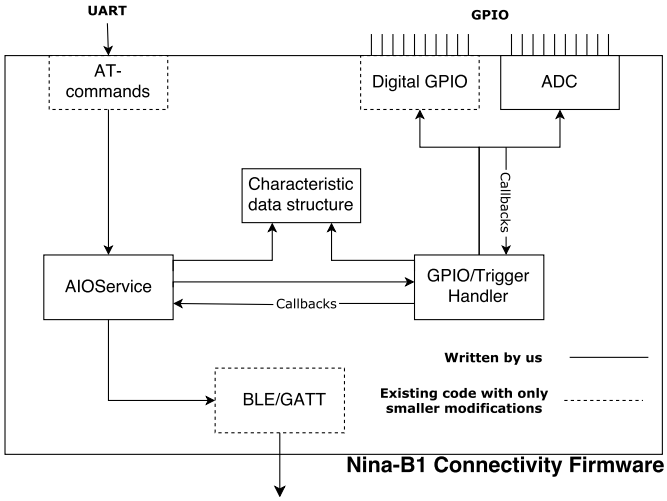

Figure 7.4: An illustration of how the different components communicates in the Automation IO Service.

Digital and Analog Characteristics

The first part of our work was to implement support for digital and analog characteristics. These characteristics are the backbone of the service, as they are what hold the actual values of the pins.

One of the first problems we encountered was that the nRF5 SDK didn't allow us to intercept a read request for a characteristic from a connected device. Instead, it automatically responds with the value stored in the GATT-table of the BLE stack. The solution we wanted to implement would read the value of the GPIOs (also referred to as pin or pins in this section) corresponding to the characteristic when a read request was received and respond with the current value. However, since this wasn't possible, we always had to keep the characteristic values up-to-date in the GATT-table. This requirement introduced some unnecessary logic where we had to periodically read the current value of the pins and update the values in the GATT-table, rather than just intercepting a read request and respond

with the current value of the pins at that time. This way of periodically polling the value of the pins would, however, turn out to be a blessing in disguise, as this behavior would become a necessity later in the project.

When a connected client writes to a characteristic, we have to update the state of the pin associated with that characteristic. In our initial implementation, a digital characteristic only contained the state of the GPIOs related to it and no information of the connection between the physical GPIOs and their respective values. Each characteristic had to include a data structure to store what physical GPIOs the particular characteristic contained, so we could find which pins to write to when a client updates the value of a characteristic. Similarly, when the value of a pin was changed, we had to find all characteristics associated with that pin. A change of a pin value can happen hundreds of times per second, meaning we have to find the corresponding characteristics for a pin just as often. To avoid iterating through the list of characteristics every time a pin value changes, we created a simple map (just an array) with pin number as keys and pointers to the components which include that pin as value. This structure was chosen to save processing power and RAM, as well as providing a quick way to find what characteristics are associated with each pin. This data structure was later found useful for triggers too and was moved out from the `AIOService` to the shared characteristic data structure, as seen in [Fig. 7.4].

As previously mentioned we began with implementing digital characteristics, as working with digital GPIOs appeared to be easier (can only have the value 1 or 0). A Digital characteristic can contain multiple digital GPIOs, and representing the value of each pin can be done with two bits. Each pin can be in one of four states, as seen in Table 7.1. The active and inactive state represents whether the pin has a high or low current on it. The tri-state is for removing the pin's influence from the rest of the circuit, and the unknown state is for when the device can't, for some reason, report the condition of a pin.

State	Value
Inactive state	0b00
Active state	0b01
Tri-state	0b10
Unknown state	0b11

Table 7.1: Pin states in a digital characteristic.

The value of the digital characteristic is padded to whole bytes in little endian order. An example of a digital characteristic with three pins could look like `0b00010001`, where the first and third pin is active and the second pin is inactive. The fourth pin does not exist and is only padded zeroes. Working with digital characteristics requires extracting/setting specific bits in its value. Bit shifting and masking make code less readable, however, it is a necessary evil. Below is an example for how to update the internal value of a digital characteristic.

```
// Sets a specific pin to inactive in a digital characteristic
..
pinIndex = 3; // Index for the pin in the digital
    characteristics pin list
mask = (~(1 << (2 * (pinIndex % 4)))); // Ones except the two
    bits associated with the pin.
value[pinIndex / 4] = value[pinIndex / 4] & mask;
...
// Now update the digital characteristic in the stack with the
     new value
sd_ble_gatts_value_set(..., ..., &value);
```

After we had finished our implementation of digital characteristics, it was easy to add support for analog characteristics. An analog characteristic can only contain one pin, and its value is represented by a 16-bit number. Having only one pin in the characteristic made it simpler to update and extract values from the characteristics since no masking was required. However, reading an analog value from an analog pin turned out to be harder than reading a digital value. An analog pin must be sampled using an ADC (Analog-to-digital converter). Functionality for ADC is included in the SDK and was wrapped by u-blox code. However, the wrapped code didn't sample with the correct speed and didn't support sampling of multiple analog signals, so we had to write our own ADC from scratch.

Aggregated characteristic

The aggregate characteristic is a characteristic that represents the state of all analog and digital characteristics with the read property, aggregated in a single characteristic. The value of the aggregate characteristic start with the state of all digital characteristics, and pads them with zeros to whole bytes, then adds the analog 16-bit values. An example of the aggregate characteristic, if the service contains the digital characteristic from before `0b00010001 <=> 0x11` and combine it with an analog characteristic value `0xA202`, the aggregate characteristic value is `0x11A202`. Remember that characteristic values are written as little endian, so the actual value of the analog characteristic is `0x02A2 <=> 674` and not `0xA202 <=> 41474`.

When there are multiple digital characteristics, it is not as simple as copying the values from each digital characteristics after each-other into the beginning of the aggregate. Digital characteristic values are padded if they don't contain a multiple of four GPIOs. This padding is unnecessary in the aggregated characteristic and should only have padding for the last byte. So digital characteristics values had to be un-padded before adding them to the aggregate.

Triggers

In Automation IO there are two kinds of triggers, called time- and value trigger. For a client to change the trigger conditions, a characteristic can have a descriptor containing the settings for each trigger. Time- and value triggers can be combined in several different compositions and would require unique logic for many

of them. Instead, we decided to focus on the triggers we, as well as our mentor at u-blox, found most valuable. We decided to place the logic for triggers in a separate component seen in [Fig. 7.4]. The `AIOService` is responsible for initiating the `trigger handler component` when an AT-Command is received for a trigger configuration. When a client changes the trigger condition, it is the responsibility of the `AIOService` to updates the `trigger handler component` with the new trigger configuration. When a GPIO changes value, and it fulfills a trigger condition, the `trigger handler component` makes a callback to the `AIOService`, which then notifies or indicates the correct characteristics.

The data structure created for storing characteristic data/configurations turned out to be useful for triggers as well. For each characteristic we have to save the corresponding time- and value triggers and some additional information about previous state and values. One value trigger condition, for example, is to trigger when an analog value goes from above to below a threshold, making us have to store the previous value for the characteristic as well.

For digital characteristics, we currently support 5/7 conditions and for analog characteristics 6/11. Adding support for the remaining triggers would be straightforward since many of the remaining triggers are just variations of triggers already implemented. We implemented the most "sophisticated" analog value trigger as the other triggers are just subsets of it. The reason behind not implementing every trigger was that we wanted to focus on other parts of the prototype, so the remainder of the conditions are left as an exercise for u-blox.

7.3.4 Automation IO client

In the early stages of our prototype, we used an application called NRF Connect, created by Nordic Semiconductor, to connect to, and monitor our service. The application can discover arbitrary services, characteristics, and descriptors on a connected device (NINA-B1).

Even though we did find this tool, we had two primary reasons that made us want to implement our own Automation IO client. First, we needed a more customized interface to be able to utilize and test Automation IO, second, we wanted it as a tool to show the usefulness of a generic app to control an Automation IO Service.

We decided to implement a simple Android application that was able to act as a Automation IO central and could interact with the service in an intuitive way. Implementing the client was straightforward and the only problems we had was with the Android BLE stack, and after reading about it, we found that we weren't the only ones having problems with it. We had days when our application just stopped working for some phones, and after a software update of the phone, the app started working again. The BLE stack has a problem when new services and characteristics are added to a peripheral when the phone is connected to a device. In those cases, Bluetooth has to be restarted on the phone for it to be able to rediscover new services and characteristics. We found some hidden methods in the Android API, which refreshes the device Bluetooth cache. However, it didn't always work as we expected.

7.4 Evaluation

With the prototype solution finished, we wanted to evaluate how good the prototype was and if we fulfilled the requirements stated by u-blox.

To verify that our implementation of the AIO Service complies with the Automation IO profile, we used the Bluetooth PTS. The PTS connects to the NINA-B1 module using a Bluetooth dongle connected to a computer and runs a comprehensive set of test cases defined by the Bluetooth SIG, intended to verify that the functionality of the service complies with the profile specification. The PTS has support for disabling tests for non-mandatory features. We disabled the features that our module doesn't support, such as some trigger conditions. To investigate how the inclusion of the Automation IO Service affect the rest of the NINA-B1 module, we conducted tests for power consumption, memory usage, and throughput.

- **Power consumption** By powering the NINA-B1 chip with an oscilloscope, it is possible to measure the power consumption of the NINA-B1 module. With the help of the oscilloscope, we could measure average energy consumption over a time interval. The tests consisted of different combinations of analog sampling time and indication/notification intervals.

- **Memory usage** Visual GDB shows the flash and RAM requirements after each compilation. To make sure those numbers were correct, we also extracted the flash and RAM usage from the generated .elf file, in a similar manner of how we did with OpenThread. The resulting difference was the same, except the difference given from the .elf file gave us an exact value in bytes, instead of rounding to whole KB. We also included the dynamic memory needed for characteristics and triggers. The RAM usage shown by the compiled program will always be allocated, even when not including the AIO Service. We calculated the dynamic memory by logging the size of every `malloc` made when adding characteristics and triggers. U-blox has implemented a heap, so we are calling a custom made `malloc`, which "allocates" memory from a static buffer. However, we will refer to it as dynamic memory, since by using it, other parts of the connectivity software will have less memory to use.

- **Throughput** One of the use cases for the connectivity software is to send data with the SPS. It is possible to use it to stream or send larger chunks of data. To see if our module interferes with the throughput of the SPS we conducted some performance tests. By using the s-center software, it is possible to perform a data pump. What the data pump does is to push as much random data as possible over serial to NINA-B1, then the NINA-B1 transmits it over the SPS to a connected device. For this experiment, we couldn't use the app we developed, because it didn't have support for the SPS, and we couldn't use the u-blox app which supports SPS because it didn't support AIO. Instead, we used NRF Connect. With it, we could subscribe to notifications for characteristics in the SPS, as well as in our AIO Service.

For a test to be valid, the results should be reproducible. The Power consumption results are the average power consumption over a 10 second period and the throughput tests are an average over a two minute period.

Every test we ran on the module was based on data from several independent measurements. Many factors can differ when performing measurements on a device, but we attempted to mitigate as many of them as possible by having a consistent lab setup, as can be seen in [Fig. 7.5]. We did not perform our measurements with any other lab setup, as this was not relevant as part of an integration test of Automation IO. Other factors such as the number of other devices which operate on the same radio bands can impact the test results and could unfortunately not be mitigated, as we had to run our tests in our office.

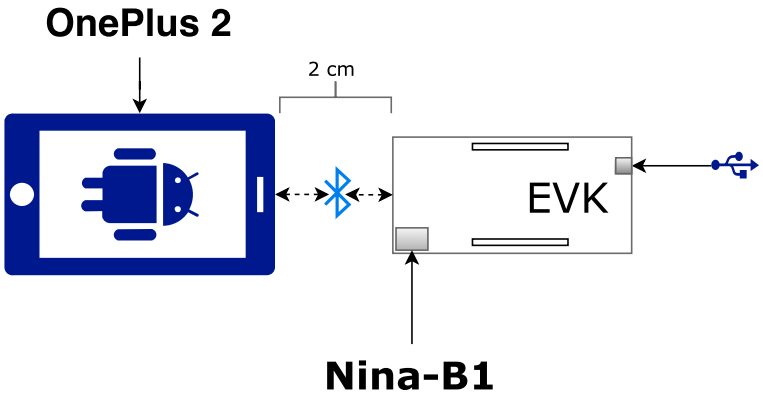

Figure 7.5: Lab setup to measure power consumption and throughput on a NINA-B1-EVK.

Chapter 8
Results

In the following chapter we will present and reflect on the result of our thesis work. We will present results for the comparison of LE technologies, conclusions about provisioning and the Automation IO profile, as well as the result of our prototype implementation.

8.1 Comparison of LE technologies

This section will display the results of the comparisons carried out in Chapter 4. We also compiled a summary of the different technologies and how they compare to each other for different metrics. This comparison can be seen in matrix form in Section 8.1.4.

8.1.1 Battery

As previously discussed in Chapter 4.1 all of the compared technologies utilize cyclic sleep patterns and the difference in battery consumption will depend on how the technologies optimize these duty cycles. Evaluating the battery consumption of the different technologies turned out to be harder than we initially anticipated. Factors such as data throughput, distance, and optimizing connection parameters all change the power consumption of a device.

Previous comparisons of Bluetooth LE and ZigBee [14][13] indicates, that when similarly configured, that BLE is $\sim 2,5$ times as energy efficient as ZigBee. It also shows that the length of the connection event, i.e. how well the power cycles are optimized, is the main contributor to the power requirements of the technologies.

Texas Instruments measured power consumption for a BLE device configured as a peripheral with a 1000ms connection interval. When running on a CR2032 coin-cell battery, their findings indicated an expected battery life of $\sim$ 400 days [26]. If the connection interval were increased to the maximum allowed value of 4000ms, the expected battery life would increase, according to the formula provided in the paper, to $\sim$ 1600 days or $\sim 4,4$ years.

We could not find a similar approximation for ZigBee, but [14] indicates that BLE has $\sim$ $2,5$ times better energy utility than ZigBee which would result in ZigBee having an approximate battery life of $\sim 1,76$ years on a CR2032 coin-cell battery.

Unfortunately, since Thread is a rather new technology, we did not find any comparisons with other low powered technologies. We did, however, find an approximation of the average battery life of a Thread end-device powered by a CR2032 coin-cell battery, waking every 10 seconds, approximating a lifetime of $\sim 2,6$ years [15].

These estimations were all for peripherals/end-devices with short duty cycles and low throughput. As these conditions change, so will the battery life of the device. These values can, however, be used as indicators for our comparison. The tests shows that BLE is the most power efficient technology which can be contributed to the four times higher data rate over-air, as well as the utilization of frequency hopping to avoid interference [14]. The greater data rate is going to lower the amount of time the device has to stay in Rx/Tx mode, thereby shortening the time of the connection events. ZigBee and Thread appear to have similar power consumption, although the limited available information on Thread makes it hard to know for sure.

8.1.2 Memory

As mentioned in Section 4.2 it is hard to compare memory usage between different wireless technologies. Variables such as compile target, library manufacturer, security features, and feature levels all factor in to the final flash size and RAM requirement of the technology. The compile target is different for all technologies, BLE and Thread are both compiled for the ARM Cortex-M family, however, not for the same models. TI's ZigBee stack is compiled for their own MCU, which is based on industry-standard 8051 MCU. The difference between the ARM MCUs shouldn't make a difference big enough to change the conclusions established in this section. The ARM and 8051 chip sets does, however, have some noticeable differences in how they manage memory. 8051 does make better use of flash and RAM, however, for bigger systems this should not impact the memory requirements in a significant way [27]. The companies responsible for development of the different stacks are all well established and we have assumed that they have well-developed stacks, without unnecessarily large memory usage.

In Table 8.1 we compare the different technologies with as similar compilation options and device roles as possible. In favor of ZigBee, we also included the non-PRO version. For Thread we only have the memory usage for a REED device, meaning the memory usage for Thread shown in Table 8.1 could be decreased further for a sleepy-end-device.

ZigBee has the highest flash requirement (121.9KB) of the three, but the best RAM (5.2KB). BLE requires about 30KB flash less than ZigBee, making it possible to fit on a MCU with 128KB flash, but application size will be very constrained. OpenThread, requiring 24KB less flash than BLE, could easily fit into a 128KB flash MCU with plenty of room for the application. RAM usage for BLE and Thread are higher than ZigBee, where BLE requires about 3KB more and Thread a whole 10KB more. However, RAM usage for a Thread sleepy-end-device could most likely be reduced significantly compared to a REED. Thread markets as a standard with low memory footprint, which match our findings for OpenThread.

We could also argue that a ZigBee router, BLE central and a Thread REED

can be compared since they have similar roles. They receive data from a low energy device and can route it to other devices. BLE does not support mesh networking; however, the central could act as a router in a piconet and could therefore perform some of the same tasks as a Thread or ZigBee router. Memory requirements for the central/router roles follow the same pattern as the peripheral/end-device roles, as can be seen in Tables [4.2, 4.1, 4.3]

Overall it looks like ZigBee requires a lot of flash memory compared to BLE and Thread, making it the worse alternative for memory constrained devices. BLE performs decently from a memory perspective and could, depending on the application size, even be run on a 128k MCU. Thread is a newly developed technology and was developed to have a small memory footprint. The OpenThread stack shows that they have succeeded in that regard, and that it is possible to have an advanced low power technology with mesh networking and still keep the memory requirements low.

Stack	Device type	Security	Flash	RAM
ZigBee PRO	End-Device	on	128.2KB	5.3KB
ZigBee	End-Device	on	121.9KB	5.2KB
BLE S110	Peripheral	on	92KB	8KB
OpenThread	REED	on	68KB	15KB

Table 8.1: Memory usage for end-devices summed from Tables 4.1, 4.2, 4.3.

8.1.3 Security

Some of the main attack surface for low powered wireless technologies in general are: Passive eavesdropping, MITM attacks, and for BLE, identity tracking. This section will compare how the different technologies handle these security threats.

- **Identity tracking** Bluetooth LE handles the problem of Identity tracking by using Identity Resolving Keys to change its public address [6, p.31] periodically. Bluetooth also features a non-discovery mode to hide the devices public address from general-discovery. Tracking devices in a Thread or ZigBee network based on signal strength is entirely possible, but as ZigBee and Thread devices are not designed to be carried around by a person, this does not impose the same threat.
- **Passive eavesdropping** Passive eavesdropping is a problem mitigated by encrypting data in transfer, as well as proper management of secure keys. BLE, ZigBee, and Thread all uses AES-CCM cryptography to encrypt their data and should be secure against passive eavesdropping, as long as their secure keys are not acquired. Each connection generates unique keys in both Thread and Bluetooth LE and as long as the key-exchange process is secure Thread and BLE keys should be safe as well. ZigBee devices have been known to come pre-installed with default network and link-layer keys with a history of having been publicly exposed [23]. As mentioned in Chapter

4.3 ZigBee also present a security flaw where it sends an unencrypted key through the network that could allow a network sniffer to acquire the key.

- **MITM attacks** How resilient a technology is to MITM attacks depends on how to secure the pairing/joining process of the technology is. For a key-exchange process to be resistant to MITM attacks, the key-exchange must use a secure procedure, as well as provide a way for the user to verify the authenticity of the connection. Thread was built with security in mind from the beginning and always require a network administrator to verify the joining device before allowing it to join the network. This joining process uses Schnorr signatures as a passkey to verify the connection. Bluetooth LE allows the user to specify what security mode and level they want for their device. Some of these security levels do not provide MITM protection while some do. If the device is running a Bluetooth version lower than 4.2, the only "secure" option is using either passkey or OOB for verification [19]. Bluetooth 4.2 introduced Secure Connections, further increasing the security of the verification methods and adding the Numeric comparison option, considered highly resilient to MITM attacks. ZigBee should, in theory, be safe to MITM attacks, as all ZigBee devices come pre-installed with keys for secure communication. However, since these keys are known to be compromised, MITM attacks are entirely possible for ZigBee.

In summary, Thread and Bluetooth LE appears to provide the strongest overall security and is proven to be resilient to both passive eavesdropping and MITM attacks. For BLE, however, the security is entirely dependent on what security mode/level, verification process, and Bluetooth version the user decides to use. ZigBee uses strong cryptographical algorithms for data in transfer, but displays some major security flaws when handling secure keys and could be vulnerable to both MITM and Passive eavesdropping attacks.

8.1.4 Summary of the technologies

In [Fig. 8.2] we have summarized the main differences between the BLE, ZigBee, and Thread technologies. We have included a summary of the metrics that have been previously discussed, as well as some other differences we have come across in our research. Some of the data for Thread has been taken from evaluations of 6LoWPAN, but as this is the technology Thread is built on, it is bound by these values.

8.2 Automation IO use cases

The purpose of this section if to find use cases for the Automation IO Service in a u-blox product, as well as for other companies looking to implement Automation IO in their own products. We will also use these use cases as a way to evaluate the different approaches presented in Chapter 5 and 6.

[1] Limited by 6LoWPAN, Thread is bound by these values.

	Bluetooth LE	**ZigBee**	**Thread**
Battery consumption	$\sim$ 4,4 years	$\sim$ 1,76 years	$\sim$ 2,6 years
Memory requirement End-device	92KB Flash 8KB RAM	121.9KB Flash 5.2KB RAM	<68KB Flash <15KB RAM
Encryption	AES-128	AES-128	AES-128
Key exchange	User defined. Support for password and Numeric comparison.	JPAKE with Juggling.	Pre-installed keys and Security Center.
Max Data Rate [6][28]	1Mbit/s	250kbps	250kbps[1]
Nominal range [14][28]	50m	10-100m	25-50m[1]
Interference handling [14]	Frequency hopping	CSMA/CA	CSMA/CA
Mesh capability	No	Yes	Yes

Table 8.2: A summary of the different low powered wireless technologies and their differences.

- A small company manufacturing watering systems for gardens wants to offer their customer the option to control their watering systems from a mobile device. It is a somewhat small business, with limited knowledge of BLE and low-level programming. The companies goal is to be able to turn on/off the watering system wirelessly, as well as notify the user if the moisture in the soil is below a certain level.

  The company decides to purchase a cable replacement module, such as a NINA-B1 module pre-flashed with support for Automation IO, and connect the GPIOs on the chip to the old system. The Automation IO Service can map the pin to control the on/off switch to a digital characteristic, and the moisture sensor to an analog characteristic. The NINA-B1 module, which has recently added support for running start-up scripts, is provisioned by flashing a script to the module. End-users can then control the system by using a mobile application built for managing generic AIO Services, developed by a third party.

  Provisioning the device through a start-up script is ideal for this use case. The watering company, with its limited programming knowledge, can provision the device through a high-level programming language and only have to learn the API for interacting with the module. A start-up script will also re-provision the device when rebooted, which is ideal when the end-user just want the product to work without provisioning the device. Depending on the model of the watering system, it is also possible for the company to customize the script depending on the needs of each model.

  If the company wanted a custom solution, rather than the Automation IO solution, they would have had to develop custom firmware, as well as a client application. For a small company with limited resources, the use of a pre-defined standard is going to save both time and money.

- A large production company using robots in their production line want to upgrade their current system architecture. Previously the robots were coor-

dinated by a master computer through physical cables, which the company wants to replace with a wireless solution. The robots need to communicate when they have finished a task, as well as additional information required to prepare the other robots on the line (Such as if the robots have to switch tools). An Automation IO module, such as the NINA-B1, could be used for this task. As the robots only have to communicate information that can be conveyed through the state of analog or digital pins, the Automation IO Service is sufficient for exposing the signals to the shared master computer. The master computer acts as a central for all connected robots and runs the logic for informing the other robots when certain tasks are finished.

The modules are provisioned over the UART connection from the module connected robot. Since each robot has a slightly different set-up of pins, they can push AT-Commands over UART to provision the module to the specifications required by each robot. A generic set-up script or hard-coded solution would not be flexible enough for this use case. The modules could also receive AT-Commands wirelessly over the included SPS from the master computer if additional provisioning were required to accommodate the rest of the system.

As Automation IO sufficiently fills the need for the company, there is no need for them to invent a custom standard and they could instead adopt the Automation IO profile. Using a pre-defined profile will not only save the company time but also reduces the risk of error when developing their product.

- In a home where several different appliances offer BLE connectivity. The homeowner wants to be able to connect to each of the home automation devices to monitor and control them. For instance, the owner wants to be able to turn on/off light bulbs as well as read the status of the washing machine with a single mobile application. If each of the home automation devices ran a custom profile to expose its data, the user application would need to include logic for how to handle each of the custom profiles. In home automation, it is common that the different devices are produced by various companies. To give the end-user a simple way to interface with all of these devices without needing a separate controller for each device, it is important that the devices agree on one protocol for communication. Automation IO can act as that protocol, as it is designed for generic IOMs and could be adapted to fit most home automation appliances. The problem arises when a home automation device wants to convey more data that can be represented simply by pins. In that case, something like the SPS would be better suited for conveying the data.

  Another benefit of using the Automation IO profile for home automation devices would be that a technician could connect to and diagnose any arbitrary device in the house through a single diagnostics device. This would make it easier for the technician to troubleshoot devices without having to find the specific connectivity software required for each device.

- A company develops a circuit where one of the output pins is connected to a i^2C circuit. An I^2C circuit is a way to connect a single output from a pin too

many input signals for other chips. This could be solved with a modified version of Automation IO by having one characteristic for each chip connected to the I^2C circuit and with an internal address for the different chips. The company decides to implement a custom version of Automation IO, where a characteristic has a descriptor for the internal addressing required. The company provisions their device by hard coding the Automation IO settings in their custom firmware, as every module has the same layout and there is no plan to change the chip in the foreseeable future.

Automation IO is a generalized standard and does not provide the ability to extend the profile with additional information needed for specific use cases. In the Automation IO specification, it is also stated that a characteristic should be directly representative of the pins it is mapped to, which would not be accurate in this use case, as severe characteristics would be needed to represent a single pin.

- Companies developing products that have an interface to the outside world, such as pins, can include logic in their product that acts on the state of those pins. The companies are looking for a method to control and monitor their products wirelessly. Instead of including the Bluetooth connectivity inside of their own product, they use a standalone cable replacement module running Automation IO and connects it to the pins of their product. The company can now either use an existing Automation IO client or create their own to control and monitor their product over Bluetooth.

Provisioning

When provisioning a device, we have shown that every approach presented in Chapter 6 can be useful. There are, however, some choices that will be preferred in most use cases. Hard-coding a solution should only be the preferred solution if the design of the IOM and desired structure of the service is previously known, there is no module connected host, and there is no room for a scripting language. The hard-coding approach leaves a lot to be desired for flexibility but does however have its benefits in terms of performance.

Running a script at start-up to provision the device is ideal if the module is not connected to a host MCU. The module can be used as a standalone module and will be ready to use directly when powered on. A module without a connected host could also be provisioned by sending AT-Commands over-air. This would, however, require this to be done every start-up, as well as require a service to receive these commands, such as a SPS.

Running a scripting engine and storing a script on the device does, however, demand both memory and processing power from the device. If the device is connected to a host, it could instead be provisioned by having the host push AT-Commands through the device UART. This approach would require less power from the module and will also offer more flexibility. A device could, if needed, add more characteristics to a module without having to reboot it, as AT-Commands are sent asynchronously over the UART.

Standard or custom solution

We have shown through our use cases that the Automation IO profile is highly flexible and can be adapted to most use cases where exposing digital and analog pins are required. We have also shown the potential benefits of using a standardized profile when exposing data, rather than building a custom profile. A standardization brings with it the benefits of providing connectivity between every device that implements the Automation IO profile. For instance, a vendor could utilize a generic AIO client application developed by a third party for monitoring their sensors. The use of a common standard is especially important if AIO is to be used in home automation, as needing to include different custom profiles for every device in a home would not be a sustainable solution. A pre-defined profile will also decrease the risk of error when implementing a service, as the specification has been tested and designed to cover every use case. From our findings, we would always recommend following the Automation IO profile when exposing I/Os over BLE if possible, as it brings with it several benefits and few disadvantages. It is, however, not always possible to follow the Automation IO specifications. Automation IO is a specification for low lever monitoring and control of pins. If a device has to include more information than what is supported in the Automation IO profile, it would not be possible to follow the specification. Also for the times when real-time communication over pins are necessary, AIO might not be a feasible solution.

8.3 Resulting product

This section will discuss the results acquired when running the tests and measurements explained in Section 7.4.

8.3.1 Automation IO proof of concept

Our final prototype of the AIO Service, running on a NINA-B1 module, supports every mandatory and many of the optional features specified in the Automation IO profile. The prototype supports all types of characteristics and most of the critical trigger conditions.

Profile verification

As previously mentioned we verified our implementation with the PTS. We modified the PTS to run every test applicable for our prototype, testing a wide variety of regular/edge cases. Some of the cases the PTS tests for correctness are:

- Characteristics are formatted correctly.
- Writing and reading to characteristics.
- All mandatory descriptors are present.
- Configure all different configurations of triggers and check that they send the correct indication/notification.
- Check that triggers only trigger when they are supposed to.

After a couple of iterations of tests and modifications of our prototype, the final product is a prototype that passes every PTS test and thus should comply with the Automation IO profile.

Power Consumption

To be able to compare the results from our different measurements, we had to establish a baseline for the power consumption of the device when running without an AIO Service. We implemented the AIO Service in such a way that if the AIO Service is not set-up by AT-Commands it will not exist on the server and therefore will not impact the power consumption of the device. This baseline test is the second result in Table 8.3, where the power consumption was 1.61 mA. When measuring our baseline, we connected a central to the NINA-B1 module but had no logic running for the AIO Service. After establishing our baseline, we measured the power consumption for various combinations of analog sample time and notification/indications. All of our measurements are the result of measuring the mean power consumption over a period of 30 seconds. The measurements were performed by having one analog characteristic, running at different sample rates and notification/indication intervals.

From studying our measurement data we can see that sampling alone does not impact the power consumption considerably, 1.61 mA compared to 1.66 mA. We can see the same pattern when examining the difference between running 10 notifications/s, but varying the sample time between 10/s and 100/s on row 4 and 6 in Table 8.3, where the difference is 0.03mA. We could also not see any noticeable difference between sending indications or notifications. In theory, indications should require more power, since indications require an ACK before sending the next indication package. Having to wait for an ACK should cause the length of a connection event to be longer, and therefore the device must be awake a longer period. The big difference in power consumption comes from the number of indications/notifications per second. Increasing the number of notifications sent from 10/s to 100/s increased the power consumption of the module by 0.3 mA.

By configuring triggers to only notify/indicate a characteristic when it is needed can save a lot of power. Avoiding the transmission of unnecessary indications/notifications is crucial if power consumption is an important factor. The best way to avoid sending unnecessary indications/notifications is to have well-constructed trigger conditions. If the value is intended, for instance, for a human to monitor a device, it might be sufficient to notify the value at most once per second.

The service disables sampling and triggers when there isn't a central connected to further save battery. Sampling and triggers are started when a central connects and stops when it disconnects. If no central is connected, there isn't any logic executed. Therefore when the Nina-B1 is idle, our AIO Service shouldn't draw any power.

Method	Samples	Power cons.
Not connected. Advertises every 1 second	off	1.57 mA
Connected, no timers running no notifications or indications	off	1.61 mA
Connected, no timers running data pump over SPS.	off	2.59 mA
Connected, 10 notifications/s	10/s	1.64 mA
Connected, 10 indications/s	10/s	1.64 mA
Connected, 10 notifications/s	100/s	1.67 mA
Connected, 100 notifications/s	100/s	1.94 mA
Connected, 100 indications/s	100/s	1.95 mA
Connected, no notifications or indications	100/s	1.66 mA

Table 8.3: Average power consumption of the NINA-B1 in different scenarios.

Throughput

Testing throughput is part of the tests to make sure we don't impact the current connectivity software. By testing the throughput of the SPS while running different configurations of AIO, we could see how much our service interfered with the SPS. We picked the SPS for our throughput tests, as the SPS is one of the most common tools for end-users of the NINA-B1 modules. When measuring throughput, we looked at how many kbps the SPS could transmit while running different configurations of the AIO Service. We also investigated a metric we call "discarded" that represents the percent of AIO notifications discarded because of having a full transmit buffer. The "discarded" metric will be further discussed later in this section. We decided to use notifications when performing our tests, as it should give the AIO Service a higher throughput and therefore should impact the SPS more. The results in Table 8.4 show the throughput of the SPS when running AIO in different configurations. The results show in Table 8.5 are intended to show the limit of how many indications/notifications can be sent without discarding notifications when running the AIO Service without any other services.

The maximum throughput of the SPS, without anything else running on the NINA-B1, was 27 kbps. When including our AIO Service, without running any logic from it, reducing the throughput of the SPS to 24 kbps, which we currently can't explain. The throughput of the SPS when including the AIO Service in an idle state is going to be the baseline for this experiment. We can, unsurprisingly, see a decrease in the throughput of the SPS while increasing the frequency of notifications. We built our prototype in a way where the notifications of the AIO Service had priority over the SPS, and since the SPS data pump is attempting to flood the channel with as many packages as possible, it is only logical to see a decrease in throughput. What we found most interesting is the difference between having three separate analog characteristics or a single aggregate characteristic. Including a way to notify the entire state of all characteristics at once was an excellent idea when designing the AIO specification. The row showing notifications/s, is per characteristic, meaning when we have three characteristics, the number of

notifications/s is three times as many as when having an aggregate. In row 3, we can see that with three characteristics and 20 * 3 total notifications/s sent, the SPS drops to 17 kbps, but with the aggregate, the SPS can still transmit at 24 kbps. Sending one notification with six bytes payload is going to be more efficient than sending three different notifications with two bytes payload each (one analog signal is represented by two bytes). With three separate analog characteristics which notify individually, the percentage of discarded notifications increase quickly. What should be noted is that these tests are edge cases. Normally the load of the SPS would not be as high as the data pump.

For the setup we were using when measuring the number of notifications that get discarded when running only the AIO Service, notifications start getting dropped somewhere between 100 and 200 individual notifications/s.

In conclusion, by using reasonable trigger settings, it should be possible to run the Automation IO Service in parallel with the SPS without any interference. A regular use case for the Automation IO Service would include having at most 10 characteristics, none of which notify more often than once every second. We can see in our results that this shouldn't decrease the throughput of the SPS at all, even when running the data pump.

	1 analog		**3 analog**		**Aggregate 3 analog**	
Sent/s	kbps	Discarded	kbps	Discarded	kbps	Discarded
0	24	0%	24	0%	24	0%
1	24	0%	24	0%	24	0%
10	24	0%	22	0%	24	0%
20	23	0%	17	0%	24	0%
50	19	0%	6	10%	18	0%
100	10	0%	5	50%	6	0%
200	3	23%	-	-	2	16%

Table 8.4: Throughput running SPS data-pump together with AIO notifications.

	1 analog	**3 analog**	**Aggregate 3 analog**
Sent/s	discarded	discarded	discarded
1	0%	0%	0%
10	0%	0%	0%
20	0%	0%	0%
50	0%	0%	0%
100	0%	30%	0%
200	6%	67%	11%

Table 8.5: The amount of discarded notifications when running only the AIO Service.

Memory requirement

The final test we performed is for the memory requirement of the AIO Service. Ideally, we would like the AIO Service to have a low memory footprint, to save memory for the other parts of the connectivity software. Some relevant data for this experiment is flash and RAM requirements for the connectivity software before and after the code for the AIO Service is added, which can be seen in Table 8.6. The dynamic memory requirement for adding characteristics and triggers are shown in Table 8.7. Reducing the dynamic memory required for characteristics and triggers could be done by replacing some arrays we implemented with a fixed size with linked list structures, which doesn't allocate unnecessary memory.

The GATT-table in the Nordic BLE stack (the SDK) has a limited size. Characteristics and descriptors take up rows in the GATT-table and filling this table up are going to be what limits how much memory our service can allocate at most. Our tests show that the limit for the AIO Service is about six characteristics with every optional descriptor and trigger for each characteristic. Calculating the maximum size of the AIO Service can be done by using the data in Table 8.7 and results in a maximum memory requirement of approximately 900 bytes. It is possible to increase the size of the GATT-table by modifying Nordics BLE stack and in that case the calculation would have to be modified accordingly.

	Original	**Automation IO**	**Difference**
Flash	406KB	422KB	16.2KB
RAM	55KB	55KB	416B

Table 8.6: Memory usage without the AIO Service enabled.

Dynamic	**Dynamic memory**
Digital Characteristic	64B
Analog Characteristic	77B
Aggregate Characteristic	72B
Per unique GPIO with one or more Characteristics	44B
Value/Time Trigger	28B

Table 8.7: Memory costs for adding more characteristics to the AIO Service.

Further improvements

The AIO Service is fully functional. However, some minor things should be discussed/updated/fixed before deploying our prototype into production.

- If the transmit buffer is full, it means that we are trying to send more data than the BLE stack can handle. Usually, it is not a problem that the transmit buffer is full, and you can just wait until there is room in the buffer. The SPS uses this technique. For AIO we have triggers that are configured

to send updates at a particular speed, or as soon as a value updates. If the BLE stack can't handle this speed, there is a question of how to manage packages that can not fit in the buffer.

We experienced this problem when we performed our throughput tests, after the implementation phase of the thesis and therefore didn't have time to find an optimal solution to this issue. What we did was to introduce a FIFO-queue with about 20 slots available, which act as a buffer for our service. When the queue is full, we throw away the top element (the oldest) and add the newest notification/indication. This method discards the oldest queued packet, but there is a problem with this approach. If one characteristic trigger very fast, and one very slow. For instance, if we have one characteristic which triggers 100 times per second, and one which triggers every 10 seconds. Losing one notification/indication from the first characteristic would not be critical, but overriding a notification from the second characteristic because of the much faster first characteristic pushing it out of the queue could cause large problems. The worst case would be that the slowly updating characteristic is never notified/indicated.

A solution could be an array with buffers for each characteristic and then cycle through it when the Tx buffer is free. This approach would give all characteristics the same priority and therefore solve the problem above.

Also, something which needs to be discussed, is which service should have the priority when notifying/indicating. Currently, both the SPS and the AIO Service sends notifications/indication to the stack individually, which is a classic race condition between the services. Extracting the logic for sending notifications/indications to a shared queuing system would be the ideal solution. Other services can then use this system to send notifications/indications.

- Since Nordics SDK doesn't allow for intercepting read requests, our solution instead periodically poll every pin and updates the current value in the GATT-table. For some use cases, this behavior is entirely unnecessary, and if a read requests could be intercepted the polling time could be optimized. For instance, if an analog characteristic had a time trigger that notifies once every 60 seconds, it would be exaggerated to sample the analog pin ten times/s, which is our current default. If we instead could sample it as the notification/indication was to be sent and when we received a read request, we could reduce the sampling frequency by as much as 99,9%. This approach could help save both processing power and reduce power consumption.

  This optimization would, however, not always be possible. For instance, if the characteristic has a value trigger, each sampled value would be significant to whether the characteristic should trigger or not, and could not be disregarded.

  This improvement could be interesting to investigate but would require that the Nordic SDK added support for intercepting read requests.

- The ADC we implemented is fully functional, but how we process the sampled value could be further improved. We do not currently support any noise

reduction of the signal and a single faulty sample could cause a trigger to send a notification/indication incorrectly. A solution to this problem would be to require the analog signal to supply a user-defined number of identical or near identical samples in a row before the state of the pin is updated in the AIO Service.

8.3.2 Android application

To fully evaluate AIO we implemented a client for Android. Since it follows the Automation IO profile, it is capable of connecting to any Automation IO server. The client does, however, not support all features of the profile and only support the ones we have implemented in our service. The reasoning for not fully implementing the profile was that we would have no way of verifying our implementation, as our server didn't have support for every feature.

Images of our application can be seen in [Fig. 8.1],[Fig. 8.2],[Fig. 8.3].

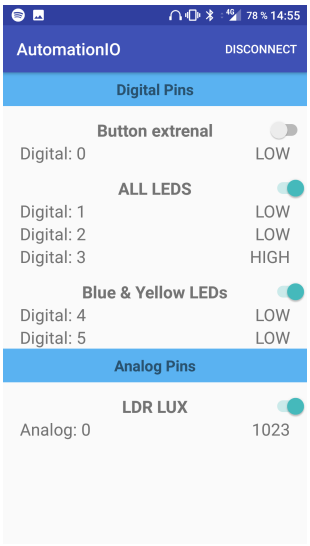

Figure 8.1: Screen showing all exposed GPIOs on the server.

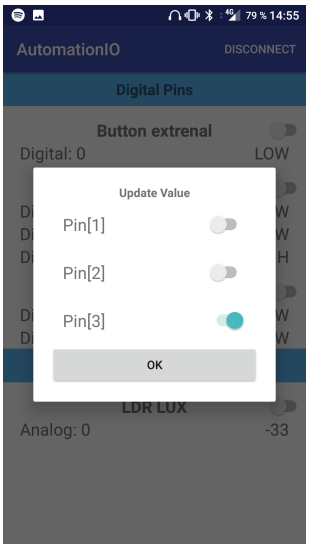

Figure 8.2: Write to individual pins in a digital characteristic.

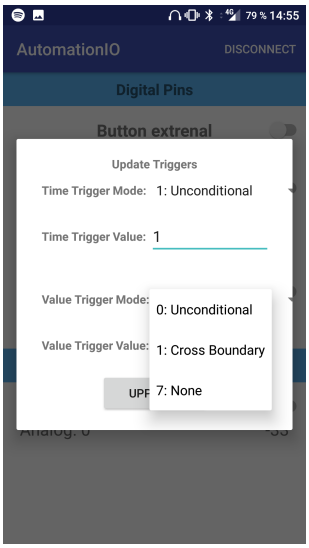

Figure 8.3: View for writing to a trigger descriptor.

The application has full support for reading/writing to characteristics, modifying any potential trigger conditions, subscribing to notifications or indications, and can sort characteristics based on whether it is an analog, digital, or aggregate characteristic.

Chapter 9
Discussion and conclusions

In this chapter, we reflect on the results presented in Chapter 8 as well as general thoughts about working with this project.

9.1 LE Technologies

Previously in this thesis, we made a comparison of three popular LE wireless technologies. We studied power consumption, memory requirements, and security, and found that the technologies are similar in many ways. They all utilize duty cycles to save power, and in regular use they should end up with a battery life of at least one year on a CR2032 coin-cell battery. Thread has the lowest memory requirements, even though we only found measurements for a REED, which would require more memory than a sleepy-end-device. A Thread device can fit into 128 KB flash and 8 KB RAM, making it a great option for limited SoCs. A BLE peripheral doesn't require much more memory and is therefore also a good choice when memory is restricted.

All technologies display similar power usage and memory requirement, however, Bluetooth LE consumes the least power and Thread requires the least memory, and sometimes this will be the decider on what technology to use.

The thesis also examined how the different technologies would handle exposing low level digital and analog signals. Since I/Os can be wired to pretty much anything, it could be catastrophic if someone unauthorized gained access to the module. This is the reason why it is highly important that the technology exposing the I/Os provides high protection against malicious entities. Each of the technologies provides encrypted communication and should, in theory, provide a secure network. However, as mentioned in Section 8.1.3, ZigBee does suffer from some security flaws due to their key management protocols and use of pre-installed network and link-layer keys. Having an educated network administrator that is aware of the flaws in the ZigBee security model is crucial. Otherwise, they might get a false sense of safety. Bluetooth and Thread both provide a security solution where the authenticity of a connection has to be confirmed by a moderator of the system, offering protection against most of the common threats against low-powered technologies. We do feel that Thread networks do provide the most interesting approach to security by giving users an easy way to decide whether a device is allowed to join a network by using a commissioning device and always

enforcing a high-security model.

Many of our findings talks against ZigBee. It has the worst power efficiency, requires the most flash memory, and has some major security flaws. ZigBee does, however, come with a couple of benefits not present in the other technologies. ZigBee provides an extensive amount of profiles for controlling household appliances and is widely used in home automation. For instance, ZigBee provides profiles for e.g. light bulbs/dimmers, thermostats, pumps, and home alarms. BLE is an attractive technology, but the lack of home automation profiles and its single-hop technology makes it harder to cover every connected device in a household. BLE do not support mesh functionality, but if the BLE central act as a Wi-Fi/Bluetooth gateway, the information can still be exposed outside of the BLE piconet and in that way create a home network. In the wearable device market, Bluetooth is currently uncontested. This is mainly because both Thread and ZigBee require stationary routers and are not designed to be mobile, as well as the widespread of Bluetooth in mobile phones.

Thread differs from BLE and ZigBee when it comes to the application layer. Thread doesn't define the application layer, making it free for the developer to choose how he/she want to implement it. Based on our evaluation of Thread, we feel it is a strong contender for the IoT. The memory requirements for Thread is great and if it is possible to fit the ZigBee application layer on top of Thread, without adding too much memory, Thread might pose a problem for the survival of ZigBee. As both Thread and ZigBee can be built on the same underlying hardware architecture and are intended to solve the same use cases, Thread might spell the end of ZigBee.

We believe that Thread with its IPv6 compatibility together with a smart application layer, e.g. the ZigBee application layer, will rule the home network. BLE does what it does well and if the use case is a wearable product or for a simple single hop use, Bluetooth wins out over the other technologies. The question is if BLE can fight its way into the home network. The problem with BLE is that a BLE product is limited to its piconet and might not be reachable from the other side of a home. If reachability between different piconets is required, a bridge with its own networking logic would be necessary, whereas in Thread and ZigBee the mesh networking works as is. Having a few ZigBee lights in a house would, for instance, extend the ZigBee network to an entire house, as each of the lights can act as a router for the rest of the network, same goes for Thread. If you add a new ZigBee product, it will, therefore, be reachable inside the entire house. Something to note is the recent release of Bluetooth 5.0 that supports mesh networking for Bluetooth Regular. Perhaps this can be used to extend the network in a simpler way for BLE devices and make BLE a competitor to ZigBee and Thread for home automation purposes.

9.2 Automation IO profile

In our investigations, we have concluded that Automation IO is a comprehensive and flexible profile and fills the gap of low-level control of IOMs that has previously been missing in BLE. Controlling the pins of any IOM provides a low-level interface

to practically any device and the use cases for such an interface is nearly endless.

We also investigated whether the Automation IO profile is always the correct choice when exposing I/Os. The problem of having a standard is that a standard don't consider exceptions and sometimes the Automation IO profile is just not enough to cover every single use case where exposing I/Os over BLE is needed. We did, however, show that following a standard comes with several benefits and very few downsides. Aside from the complications involved in developing a custom standard and the errors that can come with it, following a standard will also guarantee the interoperability with every other device using that standard. Having a standard opens up the possibility for companies to develop only the client or server side of the profile and have someone else produce the counterpart. For instance, a company can focus on developing wireless sensors and have the users themselves decide how they want to interact with them (application, Webb interface, etc.). Or the opposite, buy an Automation IO sensor and use a generic controller application or self-developed application, for monitoring and controlling their device.

When investigating the practical use cases for Automation IO, we found that there are an enormous amount of possibilities. There are, however, some use cases where Automation IO is more suitable than others. Automation IO is ideal when requiring a low-level interface to a peripheral, or when several devices need to communicate simple events to each other through a common central. Automation IO triggers are a valuable tool that allows BLE devices not to communicate more often than they have to and in that way save power. By letting the devices themselves notify the central when certain events occur, which could be crucial in, especially, automation scenarios.

9.3 Prototype at u-blox

The Automation IO prototype ended up being fully functional and supports all of the core components of the profile. When used for regular use cases, we show that our service doesn't interfere with the rest of the services of the NINA-B1. There does, however, still exist some problems with our prototype that would have to be addressed before using our solution in a final product. Firstly, u-blox has to decide if the Automation IO or SPS should have priority when sending notifications/indications. Secondly, the problem with the priority between different characteristics in our service for indications/notification. These problems should not be very hard to rectify and we have proposed simple solutions in Section 8.3.1.

We use AT-Commands for the provisioning of the service. Our investigation hints that it is not always the most efficient solution. For many cases, it is enough, but for a standalone MCU running our Automation IO Service, it is not a functional solution. For the Nina-B1, the standard way of provisioning is over AT-Commands and therefore it is ours too. We have talked about using start-up scripts to make the module standalone and not need a connected host. U-blox is currently working on a scripting solution and hopefully, in the future, such a scripting language can be used with Automation IO to enable some of the use cases we discussed in Chapter 8.2.

We think Automation IO is a perfect addition to the existing functionality of the connectivity software and we have shown that it is possible to include Automation IO into the current software without interference. We have highlighted the issues which need to be addressed before production. U-blox has shown interest in including our solution in their standard connectivity software and should hopefully be able to in one of their upcoming software updates.

9.4 Contributions

The thesis work was evenly distributed during the course of the project and we where both involved in most parts. Some areas of the implementation of the prototype were naturally divided among us, e.g. Jakob worked more with triggers and Kasper with the AT-Commands. We began our implementation with pair programming because it made it easier to learn about the code base and API. Later on, when we felt more comfortable with the existing code base, we started working on different parts in parallel to streamline the development. With that said, both of us have been involved in every component in our prototype. When writing our thesis, we worked on different sections in parallel but was both ultimately involved in every part.

9.5 Conclusion on the project

In conclusion, we both feel that this thesis has been a success. Both u-blox and ourselves are happy with our prototype solution, as well as the investigation on how u-blox benefit from including an Automation IO Service in their products. We would, however, have wanted to further polish our service, to remedy the problems previously discussed. Due to time constraints, this was not an option, but as u-blox intend to pursue Automation IO, we have full confidence that these problems will get fixed.

We also feel that we performed a thorough evaluation of the Automation IO profile and gave a general insight on what uses there might be for such a profile. Investigating the differences between the different LE technologies was a larger endeavor than we initially thought and should perhaps have been done more thoroughly if time permitted. We only included three technologies, where there exists several more. We also only skimmed the top of each technology, but did, however, feel we found enough information for the scope of this thesis.

Chapter 10
Future work

In this chapter we are going to discuss some of the ideas we have for the future of Automation IO. We will look at what will be the next step for Automation IO in the u-blox cable replacement modules, as well as what future possibilities there can be for the Automation IO profile.

10.1 Scripting language

To fully realize the potential of the Nina-B1, we want it to be able to run as a standalone product. There is currently not a way to provision the Nina-B1 if it is run standalone and any settings would have to be hard-coded. Having a start-up script in a higher level programming language could make the Nina-B1 standalone. Further, it could be used, not only as a way to provision the device, but to interact with the different modules of the connectivity software. For instance the scripting language could subscribe to events from the back-end of the Automation IO Service and use it for other purposes as well. u-blox is currently researching how to include a scripting language such as JavaScript/LUA/Python in their products.

10.2 Styling elements

One of the main use cases for Automation IO is for user control and monitoring of generic IOMs. The Automation IO profile is currently limited to exposing the exact state of the IOM and only has a single descriptor intended for a user descriptor. For instance, if an analog characteristic is representing the value of an analog thermometer, there is no way for the client to know what temperature a certain analog characteristic value corresponds to, or how it should show the value in the client.

An idea would be to either extend the Automation IO profile or define a new profile built on top of AIO, that could include an additional descriptor for styling elements in each characteristic. Such a descriptor could contain additional information about the data included in the characteristic and how to pre-process it before displaying it. Some example use cases could be:

- A temperature value should be displayed, but before displaying it, the analog value should be transformed to display degrees Celsius rather than a voltage.

A transformation formula could be defined in a descriptor to tell the client how to process the value. This could also be implemented in a generic Automation IO client today, even though it is not in the profile.

- A digital characteristic should map to a light switch and can turn it on/off. The client is a IoT application for controlling IoT devices. The styling descriptor could define that the characteristic should be displayed as a switch, rather than a 1/0 value. It could also define that the characteristic should have priority and should be displayed before any other characteristic.

10.3 Smarter clients

One of the use cases we use to highlight the usability of Automation IO is when it can be combined with a general application. Where it is possible to just plug anything to the chip running Automation IO. There are a few great applications today that is used to automate the daily life, two of them are **Tasker** and **IFTTT**. Those are very powerful applications and if they included support for Automation IO, it would open a lot of new doors. Some examples:

- A PIR motion sensor is connected to Nina-B1, the digital input pin is mapped to a digital characteristic containing a trigger with the condition "trigger when the pin goes from zero to one". In **Tasker** or **IFTTT** we then could send an email, make an API call to a server, turn on the lights or any other of the thousands of combination possible, when the sensor detects motion. All of this without writing a single line of code.
- Instead of buying expensive light sensors to Philips Hue, you can connect a light dependent resistor to an analog characteristic, whenever it's value goes above a threshold, we could set the application to turn on the light.

Changing the Automation IO profile to better fit some specific purposes will probably not be easy. Instead we can move the logic to the client application e.g. an Android application. The application will still follow the specification, only it can add additional functionality. We touched one of the enhancements in the previous section. Letting the user input a formula for how an analog value should be transformed before displaying it to the user. Our Android application is very simple, it only displays the information the AIO Service exposes, in a more readable way.

References

[1] Bluetooth Special Interest Group. *Automation IO Profile*, July 2015. Rev. 1.0.0.

[2] Bluetooth Special Interest Group. *Automation IO Service*, July 2015. Rev. 1.0.0.

[3] Bluetooth SIG. *Specification of the Bluetooth system*, June 2010. Covered Core Package version: 4.0.

[4] Mats Andersson. Use case possibilities with bluetooth low energy in iot applications. `https://www.u-blox.com/sites/default/files/products/documents/BluetoothLowEnergy-IoT-Applications_WhitePaper_%28UBX-14054580%29.pdf`. [Online; accessed 25-May-2017].

[5] Bluetooth SIG. Gatt services. `https://www.bluetooth.com/specifications/gatt/services`. [Online; accessed 25-May-2017].

[6] Kevin Townsend, Carles Cufí, Robert Davidson, et al. *Getting started with Bluetooth low energy: tools and techniques for low-power networking.* " O'Reilly Media, Inc.", 2014.

[7] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[8] Bluetooth SIG. Adopted specifications. `https://www.bluetooth.com/specifications/adopted-specifications`. [Online; accessed 19-June-2017].

[9] "Thread Group Inc". Thread commissioning. `http://threadgroup.org/Portals/0/documents/whitepapers/Thread%20Stack%20Fundamentals_v2_public.pdf`, July 2015.

[10] Jonas Olsson. 6lowpan demystified. `http://www.ti.com/lit/wp/swry013/swry013.pdf`, July 2014.

[11] Drew Gislason. *Zigbee Wireless Networking.* "Elsevier Science", August 2008.

[12] Christin Lee Joakim Lindh and Marie Hernes. Measuring bluetooth low energy power consumption. `http://www.ti.com/lit/an/swra478c/swra478c.pdf`, January 2017.

[13] Artem Dementyev, Steve Hodges, Stuart Taylor, and Josh Smith. Power consumption analysis of bluetooth low energy, zigbee, and ant sensor nodes in a cyclic sleep scenario. In *Proceedings of IEEE International Wireless Symposium (IWS)*. IEEE, April 2013.

[14] Matti Siekkinen, Markus Hiienkari, Jukka K Nurminen, and Johanna Nieminen. How low energy is bluetooth low energy? comparative measurements with zigbee/802.15. 4. In *Wireless Communications and Networking Conference Workshops (WCNCW), 2012 IEEE*, pages 232–237. IEEE, 2012.

[15] Thread Group Inc. Thread commissioning. `http://threadgroup.org/Portals/0/documents/whitepapers/Thread%20Battery-Operated%20Devices%20white%20paper_v1_public.pdf`, July 2015.

[16] Texas Instruments. Zstack-cc2530 release notes, April 2012. Rev. 2.0.

[17] Nordic Semiconductor. S110 nrf51. `http://infocenter.nordicsemi.com/pdf/S110_SDS_v2.0.pdf`, February 2015. Rev. 2.0.

[18] Nordic Semiconductor. Softdevice specification s132 softdevice. `http://infocenter.nordicsemi.com/pdf/S132_SDS_v4.0.pdf`, March 2017. Rev. 4.0.

[19] Matthew Bon. A basic introduction to ble security. `https://eewiki.net/display/Wireless/A+Basic+Introduction+to+BLE+Security`, 2016. [Online; accessed 24-April-2017].

[20] William Burr William Polk Elaine Barker, William Barker and Miles Smid. Recommendation for key management –part 1: General(revision 3). `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`, July 2013.

[21] Thread Group Inc. Thread commissioning. `http://threadgroup.org/Portals/0/documents/whitepapers/Thread%20Commissioning%20white%20paper_v2_public.pdf`, July 2015.

[22] H. Li, Z. Jia, and X. Xue. Application and analysis of zigbee security services specification. `https://doi.org/10.1109/NSWCTC.2010.261`, April 2010.

[23] Tobias Zillner. Zigbee exploited. `https://www.blackhat.com/docs/us-15/materials/us-15-Zillner-ZigBee-Exploited-The-Good-The-Bad-And-The-Ugly-wp.pdf`, August 2016.

[24] Zigbee Alliance. Zigbee alliance and thread group successfully demonstrate products running zigbee's universal language for smart devices on thread networks. `http://www.zigbee.org/zigbee-alliance-and-thread-group-successfully-demonstrate-products-running-zigbees-universal-language-for-smart-devices-on-thread-networks/`, December 2016. [Online; accessed 10-May-2017].

[25] u blox. u-blox low energy serial port service. `https://www.u-blox.com/sites/default/files/LowEnergySerialPortService_ProtocolSpec_%28UBX-16011192%29.pdf`, March 2017.

[26] Joakim Lindh Sandeep Kamath. Measuring bluetooth low energy power consumption. http://www.ti.com/lit/an/swra347a/swra347a.pdf. [Online; accessed 18-May-2017].

[27] Sillicon Labs Wade Gillham. Choosing between an 8-bit or 32-bit mcu. http://community.silabs.com/t5/Official-Blog-of-Silicon-Labs/Choosing-Between-an-8-bit-or-32-bit-MCU-Part-1/ba-p/155815, November 2015.

[28] Auday A.H. Mohamad Mahmoud Shuker Mahmoud. A study of efficient power consumption wireless communication techniques/modules for internet of things (iot) applications. http://dx.doi.org/10.4236/ait.2016.62002, April 2016.