Operating systems for resource constraint Internet of Things devices: An evaluation

Utvärdering av operativsystem för resurssnåla Internet of Things enheter

Andreas Elvstam
Daniel Nordahl
Abstract

The Internet of Things (IoT) is a large and rapidly expanding area with regards to both hardware platforms and software. When it comes to hardware platforms for the IoT some are more powerful and able to run a traditional OS like Linux, while other platforms are too constraint to do so. To ease the development within the IoT area an appropriate IoT OS is needed for these constrained hardware platforms, which can handle most of the functionality a traditional OS offer. Therefore, we evaluate IoT OSs targeted for low performance battery powered devices. In this thesis Contiki, mbed, RIOT and Zephyr are evaluated. The aim of this evaluation is to determine important IoT OS characteristics for resource constrained devices, and to highlight difficulties and experiences related to the building process of prototypes for such IoT devices. The evaluation of the IoT OSs were conducted on four types of data with regards to several measurable OS characteristics according to a criteria based evaluation method.

The evaluation resulted in a list of six IoT OS characteristics important for wireless, resource constrained and battery powered devices. Furthermore the evaluation highlights potential setbacks during the building process of a prototype system for such devices and it also explains what experiences that can be gained.

The conclusion of this thesis contributes with experience related to IoT OS prototype construction and also an evaluation result with respect to the six IoT OS characteristics for constraint battery driven devices.

Sammanfattning

Internet of Things (IoT) är ett område under omfattande utveckling, både vad det gäller hårdvara och mjukvara. När det gäller hårdvaruplattformar för IoT enheter finns det plattformar som är kraftfulla nog att execkvera ett reguljärt OS som t.ex. Linux. Andra hårdvaruplattformar är inte tillräckligt kraftfulla för att execkvera reguljära OS och för dessa plattformar finns ett behov av resurseffektiva små IoT OS. Dessa resurssnåla OS behöver kunna hantera många av de funktioner som reguljära OS erbjuder, men på ett betydligt mer effektivt sätt.


Utvärderingen resulterade i en lista av sex viktiga egenskaper för små resurs snåla IoT operativsystem. Dess resultat belyser även svårigheter och erfarenheter som framkommit under byggprocessen av prototyperna baserade på IoT OS:en.

Slutsatsen av denna rapport bidrar med erfarenheter från byggnings av prototyper IoT OS för resurssnåla enheter, samt ett utvärderingsresultat för IoT OS:en med hänsyn till de sex viktiga IoT OS egenskaperna.

Keywords: IoT, IoT OS, Contiki, mbed, RIOT, Zephyr, Evaluation, 6LoWPAN, IEEE 802.15.4, BLE
Acknowledgments

We would like to express our gratitude to our supervisor Ulrik Eklund for invaluable inputs and feedback during our work of this thesis. We would also like to thank Alex Rodzervski and Henrik Telander at Anima AB for the support and for providing us with the facilities and hardware needed in order to complete this thesis.
# Table of Contents

1. **Introduction**
   - 1.1 Research Questions ................................................. 2
   - 1.2 Thesis Scope and Limitations ..................................... 2

2. **Theoretical Background**
   - 2.1 Embedded IoT Operating Systems and Devices .................. 3
   - 2.2 Communication Protocols ........................................... 4
     - 2.2.1 Bluetooth low energy (BLE) .................................... 4
     - 2.2.2 6LoWPAN .......................................................... 4
     - 2.2.3 Constrained Application Protocol (CoAP) ..................... 5
     - 2.2.4 Routing Protocol for Low power and Lossy Networks (RPL) .... 5
   - 2.3 Power Analysis of embedded systems ............................ 5
   - 2.4 Modular programming .............................................. 6

3. **Related Work**
   - 3.1 A gap analysis of Internet-of-Things platforms ................ 7
   - 3.2 Operating Systems for Low-End Devices in the Internet of Things: a Survey ....................................................... 7
   - 3.3 Operating Systems for IoT Devices: A Critical Survey ........... 9

4. **Method**
   - 4.1 Stating the research problem ..................................... 10
   - 4.2 Literature Survey .................................................... 10
   - 4.3 Criteria Based Evaluation ........................................ 10
     - 4.3.1 Fixing the Requirements ....................................... 11
     - 4.3.2 Preparation of Evaluation ..................................... 11
     - 4.3.3 The Evaluation Step ........................................... 12
   - 4.4 Constructing the Prototypes ...................................... 12
     - 4.4.1 Develop a system architecture ............................... 13
     - 4.4.2 Analyse and design the system ............................... 13
     - 4.4.3 Build the prototype ........................................... 13

5. **Results and Analysis**
   - 5.1 IoT OS Requirements ............................................... 14
   - 5.2 Prototype Use-Case ................................................ 15
   - 5.3 Prototype Architecture ............................................ 15
   - 5.4 Contiki ............................................................. 16
   - 5.5 mbed .............................................................. 18
G.1 Node router connection .................................................. G-1
G.2 Ping node over IPv6 ..................................................... G-2
G.3 CoAP POST request ...................................................... G-3
G.4 CoAP GET request ....................................................... G-4
G.5 BLE advertisement ..................................................... G-5
G.6 Contiki current measurement ........................................ G-6

H mbed Test Cases .......................................................... H-1
H.1 BLE advertisement ..................................................... H-1
### Definitions and Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6LBR</td>
<td>6LoWPAN Border Router</td>
</tr>
<tr>
<td>6LoWPAN</td>
<td>IPv6 over Low power Wireless Personal Area Networks</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BLE</td>
<td>Bluetooth Low Energy, low energy implementation of the Bluetooth protocol.</td>
</tr>
<tr>
<td>BT</td>
<td>Bluetooth, a wireless communication protocol.</td>
</tr>
<tr>
<td>CMSIS</td>
<td>Cortex Microcontroller Software Interface Standard, is a hardware abstraction layer for the ARM Cortex-M series of processors.</td>
</tr>
<tr>
<td>CoAP</td>
<td>Constrained Application Protocol, an application layer protocol designed for highly constrained devices.</td>
</tr>
<tr>
<td>HDK</td>
<td>Hardware Development Kit</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IoT</td>
<td>The Internet of Things, is a concept of which billions of ordinary devices, objects, vehicles and even animals and humans are outfitted with tiny, built-in sensors and processors, allowing them to communicate to the world through the Internet.</td>
</tr>
<tr>
<td>IS</td>
<td>Information System</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit, is a single-chip computer containing CPU, RAM, ROM or flash, and often programmable I/O ports.</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit, a hardware component dedicated to efficiently handle caching, memory allocation, de-allocation, virtual memory and memory protection.</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System, is a system software designed for handling hardware and software resources for applications on a system.</td>
</tr>
<tr>
<td>RDC</td>
<td>Radio Duty Cycling</td>
</tr>
<tr>
<td>RPL</td>
<td>Routing Protocol for Low power and Lossy Networks</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System, an operating system designed to allow precisely timed scheduling.</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SLIP</td>
<td>Serial Line IP</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>UUID</td>
<td>Universally Unique Identifier</td>
</tr>
</tbody>
</table>
1 Introduction

The Internet of Things (IoT) has the potential to greatly impact the way we live our lives, with many areas where there are possibilities of connecting devices and sensors. One example is home automation, where appliances can be connected, controlled or monitored remotely, even taking decisions on their own based on information available to them. Another example is connected cars, that allows the cars to help keep track of traffic congestion and road work, and to notify other connected vehicles about these obstacles. IoT is also expected to greatly improve efficiency within the health care area [1], e.g. where doctors can conduct health checkups on their patients remotely. Another area where IoT is expected to have a big impact is production, where automation and optimization of factories and transportation can save both money, time and lower the environmental footprint. This can be achieved through automated communication between self-managing inventories, self-driving forklifts and self-aware maintenance systems [Ibid.]. The IoT market is under rapid expansion and Ericsson expects that there will be approximately 26 billion connected devices by the year of 2020 [2]. This expansion is fueled by the fact that Internet connections are getting faster, cheaper and more widely available [3]. At the same time sensors and microcontrollers (MCUs) are decreasing in both size, cost and power consumption [4] [5]. All of these trends creates the perfect environment for the growth of IoT.

This expansion has resulted in a vast number of options when it comes to hardware (HW) platforms targeted for the IoT market. To ease large scale development, maintenance, deployment and porting of applications, an appropriate OS is needed which is able to run on many different HW platforms. For high-end IoT devices there already exist well established and developed OSs like Linux [6]. The paper by Hahm O et al. [7] defines some important characteristics of an OS for small IoT devices;

• Small Memory footprint; small IoT devices have limited memory available, therefore the memory footprint of the OS needs to be kept small.
• Energy Efficiency; for battery powered IoT devices efficient energy usage is crucial.
• Support for Heterogeneous Hardware; there is a large variety of HW platforms targeted for IoT devices, and a key challenge for an IoT OS is to support these platforms.
• Real-Time Capabilities; many IoT applications require precise timing and therefore an IoT OS needs real-time capabilities.
• Network Connectivity; is one of the most important functions of an IoT OS. There exists a number of different connectivity solutions and preferably an IoT OS should support most of them.
• Security; since IoT devices are connected to the Internet, security becomes a major concern, especially for IoT devices used within sensitive areas such as medical applications.

The recent growth within the IoT field has lead to the development of numerous new OSs targeted for low-end IoT devices. These new OSs aims to offer much of the functionality that Linux offers but with a considerably smaller memory footprint, support for multiple connectivity solutions and high focus on energy efficiency.
1.1 Research Questions

As stated in section 1.1, there exists a range of different OSs and HW platforms. This variety in OSs and HW can lead to difficulties for developers as to which OS is best suited for particular use-cases. To evaluate suitable IoT OSs for future lightweight IoT devices, the research aim of this thesis is:

Theoretical and empirical evaluation of IoT OSs for low performance, battery driven HW platforms.

The empirical and theoretical evaluation is conducted in order to answer the following research questions:

**RQ 1**  What OS characteristics are important when choosing an IoT OS for prototyping of an IoT system consisting of resource constrained battery driven devices?

**RQ 1.1** What current IoT OS best meets the requirements?

**RQ 2** What difficulties related to an IoT OS target for wireless resource constrained devices can arise during the building process of a prototype?

**RQ 2.1** What experiences related to an IoT OS can be gained from the difficulties identified during the building process of a prototype for wireless resource constrained devices?

The evaluation is based on the requirements specified in collaboration with Anima Connected. Anima is a newly started company operating in the IoT space, and like many other companies have an interest in investigating what IoT OSs are suited for future products. Hence, an evaluation with regards to specific characteristics is needed. Key aspects of the evaluation are good communication protocol support as well as support for multiple hardware platforms and low power consumption. Section 5.1 contains the list of the criteria for OS evaluation, produced in collaboration with Anima.

1.2 Thesis Scope and Limitations

As stated in section 1.1, the aim of this thesis is an evaluation of IoT OSs for low performance, battery driven HW platforms. However, the theoretical evaluation is limited to four IoT OSs. The four IoT OSs evaluated are selected in collaboration with Anima based on available IoT OSs at the start of this thesis work. Only existing features of the OSs are evaluated, hence this thesis does not include porting or implementation of any features. Security is indeed of high importance within IoT, but this thesis does not cover security aspects of the evaluated IoT OSs.

In IoT mesh-network routing is essential and the IoT OSs investigated in this thesis does have functionalities for handling mesh-routing, but it is not within the scope of this thesis to conduct any deeper examinations of how the routing is handled by the OSs. Data for the evaluation of the IoT OSs is collected in several steps, and based on the data collected in the previous step, a decision is made regarding the potential of OSs. If the collected data indicates that an OS does not fulfill the IoT OS requirements specified in section 5.1, no further data is collected for that specific IoT OS. The prototypes are built in order to collect data for the evaluation, and are based on a use-case within home automation. This use-case consists of several battery powered nodes in a low-power wireless IP based network, and is explained in further detail in section 5.2.
2 Theoretical Background

This section covers the theoretical background related to this thesis. It aims to provide additional information for the reader. The aim is not to cover whole areas in depth, but rather give insight to specific areas relevant to this thesis. The areas covered in this section exemplifies characteristics and methods relevant to resource constrained IoT devices and OSs. A background related to this is relevant since it is important to consider the restrictions and limitations of both HW and OS when examining and evaluating IoT OSs.

2.1 Embedded IoT Operating Systems and Devices

There are a wide range of different MCUs on the market today, from simple 8-bit up to advanced 32-bit architectures. Building applications for these MCUs can be a time consuming task due to the many different architectures. This creates a need for an operating system or an IoT OS, in which applications can be installed and executed. Developing applications for a specified IoT OS can have benefits since most IoT OSs have built in support for wireless communication, some even offers APIs for easier power management [8]. One crucial function of an IoT OS is hardware abstraction, this makes porting of an application easier, since the application will not be HW specific. An IoT OS is equivalent to an RTOS in many ways, it is designed to be compact, energy efficient and able to run on a system with limited resources. However, a difference between a conventional RTOS and an IoT OS is that an IoT OS must have support for some type of Internet connectivity solution. Linux is a well-adopted OS and already offers most of this functionality. However, its kernel size and CPU requirements prevents it from being used on HW platforms with limited resources [9, 10]. Generally, IoT devices can be divided into two categories, low-end devices and high-end devices. The high-end devices are devices that are able to run OSs like Linux, while low-end devices are too constrained to do so [7]. Within the low-end category, according to the Internet Engineering Task Force (IETF) there are three types of specifications for IoT devices with limited resources, class 0 (C0), class 1 (C1) and class 2 (C2) [11]:

- C0 consists of devices with $< 10$ kB RAM and $< 100$ kB flash.
- C1 consists of devices with $\sim 10$ kB RAM and $\sim 100$ kB flash.
- C2 consists of devices with $\sim 50$ kB RAM and $\sim 250$ kB flash.

Devices within the C0 classification usually consist of an MCU built on an 8-bit architecture, while C1 and C2 devices can range from 8-bit up to 32-bit architectures. Even though some devices offer 32-bit processors, the common denominator of all devices within the C0 to C2 classes are that their resources are too limited to be able to run a resource demanding OS like Linux [7]. IoT devices are not expected to follow Moore’s law like conventional computers, but rather develop in the direction of higher efficiency, smaller size and lower production costs [5]. Therefore, a lightweight OS is required, an OS that can run on HW platforms with limited resources. Consequently, there have emerged a plethora of different lightweight OSs [12, 13]. Some of them are open source projects while others are proprietary. Their architectures sometimes differ but they have one thing in common, they all offer a method of connecting devices to the Internet.
2.2 Communication Protocols

On top of the existence of a plethora of IoT OSs there are also a range of different protocols [14] for IoT devices to interconnect with. Some of these protocols are widely adopted and standardized, while other protocols are proprietary. Different IoT OSs have support for various communication protocols, some support more than one while others might rely solely on one protocol. Since the supported protocols of an OS determines in what ways the device running the OS can communicate, the supported protocols of the OS is of importance when selecting what OS is suited for a specific set of devices.

2.2.1 Bluetooth low energy (BLE)

For IoT devices there exists a demand for low-power radio communication. Bluetooth Low Energy (BLE) offers a solution to low-power communication between IoT devices. Building on the global Bluetooth standard, BLE further develop the technology in terms of energy efficiency. When using standard Bluetooth for radio connectivity a connection is continuously maintained between the connected devices. BLE, in contrast only creates a connection when data needs to be transmitted or received, and then closes the connection when the data has been sent [15]. To further reduce the power consumption BLE also has a substantially lower data throughput compared to standard Bluetooth. This, together with other design choices leads to low energy consumption [16]. With the recent development of IoT efforts have been made to make BLE support IPv6 based network communication. This would enable nodes in a sensor network to send IP packages through a border router to any device connected to the Internet. Since BLE is a commonly used technology in smartphones, the smartphones can act as border routers and therefore eliminate the need of a dedicated border router.

2.2.2 6LoWPAN

With IoT, the number of devices connected to the Internet is expected to dramatically increase, this makes the use of the IPv4 protocol impractical, since it supports a limited number of IP-addresses. The new Internet protocol standard IPv6 solves this problem, offering a seemingly unlimited amount of IP-addresses. However, due to its large header size and heavy reliance on multicast, IPv6 is not well suited for low power IoT mesh networks [17]. For battery powered, small IoT devices, energy efficiency is more important than the ability to transfer large amounts of data. The purpose of 6LoWPAN is to act as an adaption layer between the IP-layer and the physical MAC-layer, see figure 1, to make data transmission energy efficient, while still taking advantage of 30+ years of IP technology development [18]. This is achieved by compressing IPv6 headers and using a more energy efficient routing protocol within the 6LoWPAN network [17]. It is designed to allow packets to be sent and received over IEEE 804.15.4, which is a solution for low data-rate and low-power consumption [19]. A major advantage of 6LoWPAN is that since it is IP based, a 6LoWPAN device can be connected to the Internet via a 6LoWPAN border router (6LBR) [18].

![Figure 1: The 6LoWPAN stack](image)
2.2.3 Constrained Application Protocol (CoAP)

CoAP is a transfer protocol designed for constrained devices and networks [20]. It uses the REST architecture and offers similar functionality to HTTP, allowing access to resources using URI’s. To achieve less complexity and using less resources CoAP has a smaller packet header and uses UDP instead of TCP. Additionally, response codes are also encoded into one single byte to further reduce the packet size. These aspects, together with other design choices makes CoAP a good fit for small and constrained network devices while still offering the functionality of HTTP.

2.2.4 Routing Protocol for Low power and Lossy Networks (RPL)

With IoT and specially within battery powered use-cases the need for routing packets to and from different devices is of essence. Many of the use-cases within IoT is also non-static, which means that the nodes in a network can move around. In a traditional Wi-Fi network the nodes (laptops and smartphones) are all connected directly to the access point which means that the access point only has to keep track of the devices connected to it. However, within IoT and for resource constrained battery driven devices the range of the networking communication of the nodes might require the network to be able to handle mesh networking. This means that the nodes might have to depend on nearby nodes to forward packets to the destination since the node might be out of range to communicate directly. RPL is a routing protocol operating on top of IEEE 802.15.4 and is designed for low-power wireless networks. The basics of RPL is that each of the nodes keep track of how far away, in terms of forwarding hops, the node is from the access point. Each node also keeps track of the neighboring nodes and how far away the neighbouring nodes are to the access point. So in order to send packets the node only needs to send the packet to the neighbouring node which is closest to the access point.

2.3 Power Analysis of embedded systems

To be able to design an embedded system powered by battery the knowledge of how much power the systems uses is of essence. The meaning of power consumption is often referred to both power consumption and energy consumption. However, in the context of battery powered devices the knowledge of a system’s energy consumption is of importance. This is due to the fact that the more energy a system consumes the faster the battery of the device is depleted, and thus decreasing the battery life of the device. To further elucidate this difference the correlation between power consumption (P) and energy consumption (E) is shown in equation 1 and 2 [21].

\[ P = I \times V \]  
\[ E = P \times T \]

Where, \( I \) = average current, \( V \) = supply voltage and \( T \) = execution time. This shows that energy consumption indeed is correlated to the power consumption of a system, but also that the energy consumption is dependant on the execution time. To give an example, a standard coin cell battery has an operating voltage of 3 V and the capacity of about 300 mAh. If a systems runs on 3 V and draws 30 \( \mu \)A the systems would be able to run for \( \frac{300 \text{mAh}}{30 \times 10^{-6} \text{mA}} = 10000 \) hours (roughly one year). If the same system could draw only 10 \( \mu \)A the same battery would last \( \frac{300 \text{mAh}}{10 \times 10^{-6} \text{mA}} = 30000 \) hours (roughly three years).
An embedded system rarely draws a consistent amount of current since the energy usage varies depending if the system is idle or executing tasks. For example, many embedded systems enters power saving modes when there is no task to execute. The energy a system uses differs a lot between when the system is idle and when the system is executing tasks. Furthermore, different tasks uses different amounts of energy. This makes the overall energy usage of a system highly dependant on what tasks the system is performing, along with how often and for how long the system is executing the tasks.

To be able to determine how much energy a system is using the system can be either simulated or measured. A simple way to measure a system’s energy consumption is by using a digital multimeter (DMM) [22]. This way the total current of the system can be observed. However, since the energy usage varies depending on different tasks, this measurement will vary during the different stages of the systems program. One way to compensate for this source of error is to isolate different parts of the program and measure how the system performs during these parts. To achieve this, simple loops of the important tasks can be created. The system can then be measured while only executing one type of task in order to observe how much energy that task in particular uses [21]. By measuring the different tasks the system will execute along with the current drawn in idle mode the average current can then be calculated.

### 2.4 Modular programming

Modular programming is a design aspect of programming where the objective is to divide a program into different modules [23]. Each of these modules are specifically designed to perform a certain task or activity and nothing else. By abiding to this design aspect a program can easily be adopted to different use-cases. Adoption is made easy by simply removing or adding specific modules containing the desired functionality.
3 Related Work

This section describes previous work related to this thesis. Such work consists of previous evaluations and analyses of IoT OSs. Work mentioned in this section is relevant to this thesis by providing theoretical background to the IoT segment along with guidelines and suggestions regarding IoT OSs and their desired characteristics.

3.1 A gap analysis of Internet-of-Things platforms

Minerauda et al. [13] conducted a gap analysis of 39 existing IoT platforms. Their aim was to list the characteristics of each platform and evaluate how well each platform meets the expected criteria. Furthermore, the authors suggest how the existing gaps should be minimized. The importance of different criteria is based upon a survey of the partners of the Finnish Internet of Things program. Apart from listing all the platforms and how well they meet each criteria, the authors also lists a summary of their gap analysis. This summary highlights the six most important areas: Support of heterogeneous devices, Data ownership, Data fusion & sharing, Developer support, Ecosystem formation, and IoT marketplace. Within each area the following five key points are pointed out: Current status, Expectations, Gaps, Problems, and Recommendations.

The paper is relevant to this thesis by contributing viable information regarding the current IoT segment. It provides background related to this thesis by explaining important characteristics of IoT platforms. Furthermore, the paper provides information regarding common shortcomings of the platforms, along with recommendation’s regarding how to overcome these shortcomings.

3.2 Operating Systems for Low-End Devices in the Internet of Things: a Survey

The objective of the survey by Hahm et al. [7] is to identify light-weight OSs for the IoT, which have the potential to become the “new Linux” for small IoT devices. To become the Linux for small IoT devices they want the OS to be open source with a large platform support. They conclude that IoT devices can be divided into two general categories based on their performance: high-end IoT devices and low-end IoT devices. The high-end IoT devices are listed as smartphones and powerful single-board computers, i.e. Raspberry Pi. These devices can often run Linux and for them there is no particular need for a new OS. However, Linux has a memory footprint that is too large for it to run on low-end IoT devices. Typical low-end devices, according to the authors, includes Arduino [24], Econotag [25] and Zolertia Z1 [26]. The authors’ focus is on the second category of devices, since they identified a need for an OS for these types of devices. To investigate different OSs the authors focused on the following characteristics of the OSs:

- Small Memory Footprint: As stated by the authors IoT devices in general have less resources than conventional computers, therefore one requirement for a generic IoT OS is to be able to function on small devices.
- Support for Heterogeneous Hardware: A key challenge for a generic IoT OS is the ability to run on different HW platforms. MCU’s from 8-bit up to 64-bits are available on the market and on top of that there exists a wide range of communication protocols. Ideally a generic IoT OS should have support for all of these different HW platforms and protocols.
- Energy Efficiency: Under this category the authors list two major challenges. Many IoT devices operate on battery and are required to operate several years on one charge. The sheer volume of expected IoT devices (the authors expects tens of billion deployed devices) requires them to be energy efficient on a global level.
Network Connectivity: The main point of IoT devices is to connect them to the Internet. Therefore a range of different communication protocols is required. The authors list Linux as an example of future proof design, and from the evolution of Linux it has become clear that its desirable to support multiple network stacks.

Real-Time Capabilities: Precision timing is listed as an important characteristic for an IoT device, with pacemakers, vehicular ad-hoc networks mentioned as examples.

Security: IoT devices can be used in infrastructural or industrial systems with life safety implications. Since they are also connected to the Internet, this expose them to serious security challenges. Therefore good security support is a key requirement according to the authors.

The conclusion of the paper is that there are mainly three design categories of OSs that has the potential to become “the Linux” for IoT. The first category is multithreaded OSs, and here RIOT is identified as the most promising one. The second category is event-driven OSs, where Contiki is proposed as the best choice. The final category is RTOSs, here FreeRTOS is selected. The final conclusion is that there is a plethora of different OSs for IoT devices, allowing the user to choose the one which is best suited for a given situation.

The paper is valuable to the thesis since it offers a definition to what features a IoT OSs is expected to support. It also offers a valuable insight in IoT OS design.
3.3 Operating Systems for IoT Devices: A Critical Survey

The objective of the paper by Gaur, P and Tahiliani, M [6] is to outline the essential features of an IoT OS targeted at small and highly constrained devices. These essential features are presented in a generic model which the authors have based upon a survey conducted on current IoT OSs and how these OSs comply with the stated features. The features deemed essential to the authors are:

- **Architecture:** There are several kernel architectures an OS can be based on and different architectures offer both advantages and disadvantages when it comes to memory footprint, performance and complexity.
- **Programming Model:** The Programming model of an OS plays an important role in both performance and productivity. The programming model should aid in increasing productivity for developers as well as utilizing an abstraction to the underlying system.
- **Scheduling:** The scheduling strategy of an OS is a key factor for the system’s performance. Some IoT systems might have strict real-time constrains and an IoT OS must be able to provide the scheduling needed for these types of systems.
- **Networking:** Since Internet connectivity is the key aspect of IoT networking is of high importance for an IoT OS. However, traditional IP-stacks are too power consuming or too complex for small highly constrained devices. Therefore, an IoT OS must support light-weight and reliable network stacks.
- **Memory Management:** The focus of IoT devices is to enable small and heavily constrained MCUs to interconnect through Internet. Many of these small MCUs do not have dedicated MMUs to handle caching, memory allocation and memory protection. Therefore, an IoT OS needs to provide efficient memory management in order to abstract the memory management from the programmer.
- **Portability:** IoT includes a vast variety of different use-cases and hence there will be a wide range of different applications. These applications will run on many different HW platforms and therefore an IoT OS should easily be portable to different HW platforms.

The survey conducted by the authors includes 13 small OSs designed for IoT devices, including Contiki, TinyOS, LiteOS and RIOT. To display how these OSs meet the six features listed above the authors constructed a table containing each OS and the six features. Furthermore, the authors also analysed the features and proposed a generic model for what an IoT OS should provide; To be able to adopt to the vast majority of applications and HW platforms a micro-kernel architecture is the most promising approach. To further specify, a modular micro-kernel approach is deemed the most promising because it allows for flexible and adoptable systems by enabling the usage of modules. Modules can easily be added or removed from applications in order to meet the specific requirements from use-case to use-case. To ease the productivity of IoT applications standard APIs and the support for well adopted programing languages, i.e. C, is a key factor. The key factor for large-scale networks within IoT is the support for IPv6. Hence, an IoT OS should support a light-weight modular protocol stack including UDP, TCP, IPv4, IPv6 etc.

The paper is relevant to this thesis by contributing key features of an IoT OS along with proposing a generic model for how these features should be implemented. By doing this the paper provides valuable information regarding the framework of this thesis along with important characteristics used during the evaluation conducted in this thesis.
4 Method

Since the workflow of this thesis includes a range of different activities it was unsuitable to use one single research method. The research workflow, shown in figure 2, consisted of three main steps conducted in the following order; 1 Stating the research problem, 2 Literature survey and 3 Criteria based evaluation. A literature survey was conducted to gain knowledge in the IoT area and the evaluation of the IoT OSs was done using a criteria based evaluation method, which is efficient when evaluating a system on measurables.

Figure 2: Research workflow

4.1 Stating the research problem

The first step of this thesis was to form the research questions. The research questions, presented in section 1.1 was formed to reflect a request from Anima to evaluate IoT OSs for resource constrained battery driven devices. After stating the research question, the scopes and limitations for this thesis was decided.

4.2 Literature Survey

To gain knowledge about the problem domain a literature survey was performed with focus on IoT OSs targeted for resource constrained battery driven devices. In section 1 we introduced the background and the framework of the problem domain. By doing this we elucidated the relevance of this thesis. Furthermore, the literature survey provided us with theoretical background related to the framework of the problem domain, presented in section 2. Through the literature survey we also gained knowledge regarding what others have done within the area of resource constrained battery driven IoT devices, presented in section 3.

4.3 Criteria Based Evaluation

The evaluation was done using a criteria based evaluation method, which is described in detail in the paper by Gediga et al. [27]. The reason for choosing this method is that it provides a framework as how to perform an evaluation based on measurable system characteristics on SW systems. This evaluation was conducted in three stages, described in the three following subsections. The goal of the evaluation was a theoretical and empirical evaluation of future-proof, IoT OSs for low performance, battery driven HW platforms.
4.3.1 Fixing the Requirements

The first stage was to identify the IoT OS characteristics the evaluation should be based on. To get an impression on Anima’s view of the IoT field and to gain knowledge on what functionality Anima sees as important for their IoT products, a discussion was conducted with Sarandis Kalogeropoulos, one of Anima’s founders. His opinions were used as a guideline on which areas of the IoT field that are important to Anima, regarding their future products. To produce a more specific list of IoT OS requirements four interviews regarding the IoT OS requirements was conducted with Alex Rodzevski, a Senior SW architect at Anima. An additional interview was conducted with Henrik Telander, Director of HW Development at Anima. The interviews and discussion resulted in the requirements listed in section 5.1 which is the foundation of the evaluation. Furthermore, the interviews also resulted in the selection of the following four OSs for the evaluation; Contiki, mbed, RIOT and Zephyr.

After specifying what criteria the evaluation should be based on the next step was to conclude what types of data that were to be obtained. Here, we deemed that it was important to make sure that the data covered all criteria included in the evaluation and the following four data types were defined; Manufacturer data, Written Interview data, Minimal Application and Prototype.

4.3.2 Preparation of Evaluation

The next stage of the evaluation was to collect the data specified in section 4.3.1:

- **Manufacturer data;** was gathered through OS developer specifications and the objective was to show what Communication protocols, Communication radio and Hardware the OSs supports.

- **Written Interview data;** was acquired by conducting an architectural analysis [28] by sending out questionnaires to developers working at companies, that are using the IoT OSs. This was done in order to study the fit of an artifact (IoT OS) into a technical IS architecture (IoT smart home). The questions in the questionnaires, which are presented in appendix D, were formed in order to reflect the requirements list.

- **Minimal Application experience;** was obtained by conducting a case study, which consisted in setting up the development environment and writing a "hello-world" application for the IoT OS. The objective of this data was to show the programming models of the OSs and how developer friendly the OSs were. To show this, our experience from creating the minimal application is observed and described.

- **Prototype construction experience;** This data was gathered by designing and building a prototype based on a use-case within home automation. The use-case was designed in collaboration with Anima, and the prototype was used to showcase a data source where all the requirements could be observed. Observing the requirements was done by describing our experience and evaluating the experienced functionalities of the OS during the building process. Additionally, black box testing were performed in order to measure the prototypes current usage and protocol support.

The data was collected and evaluated in the steps according to figure 3. After each step a decision was made if the data collected in the step showed that the IoT OS had potential to meet the requirements. The first decision was made after the Manufacturer Data was collected, and this decision resulted in that no more data were to be collected for Zephyr. The second decision was based on the Minimal Application data and resulted in that all three remaining OSs still showed potential and that Written Interview data were to be collected for all remaining OSs. However, we were unable to find any companies that use mbed in their products and as a result no data was obtained for mbed in the third data collection step. After the Written Interview data was collected the decision was made to build prototypes using Contiki and mbed. Building the prototypes was done by using a five-stage method proposed by Nunamaker and Chen [29]. The process of building the prototypes is described in section 4.4.

All data is presented in section 5 according to the alphabetical order of the IoT OSs: Contiki section 5.4, mbed section 5.5, RIOT section 5.6 and Zephyr section 5.7. In each of the sections for the OSs the data is presented in the order the data was obtained, according to figure 3.
4.3.3 The Evaluation Step

The final stage of the criteria based evaluation was to evaluate the obtained data from section 4.3.2 with regards to the requirements specified in the section 4.3.1. This evaluation was conducted for each of the OSs and resulted in a summary of the OSs ability to meet the requirements specified in section 5.1.

4.4 Constructing the Prototypes

To ensure the validity of the two prototypes they were developed according to a well established scientific method for systems development within information systems. This method consists of a five-stage development process described in the paper by Nunamaker and Chen [29]. Since both prototypes were built from the same use-case the first two stages Construct a conceptual framework and Develop a system architecture of the development process were identical. Furthermore, the stage Construct a conceptual framework was already performed in section 4.3.1. Additionally, the last stage Observe and evaluate the system is included in the Criteria Based Evaluation’s last step. The three remaining stages are covered in the following subsections, where stages Analyze and design the system and Build the prototype system were performed twice. Figure 4 illustrates the stages performed while building the two prototypes.

Figure 3: Criteria based evaluation workflow.

Figure 4: Building process for the two prototypes
4.4.1 Develop a system architecture

In this stage an architecture of the prototype was developed in collaboration with Anima’s representative Alex Rodzevski. This was done in two one hour long meetings where the use-case of the prototype was discussed and with respect to the use-case, presented in section 5.2, a prototype architecture could be formed, presented in section 5.3.

4.4.2 Analyse and design the system

In this stage the architecture of the system was analysed. This was done in order to adapt and design the use-case to the two IoT OSs selected for prototype development. The architecture of the use-case was divided into modules, according to figure 5, which could be developed independently. Adapting the use-case was done twice, once per OS, and consisted of selecting the communication protocols and radio for the low-power wireless network. After the selection of connectivity solution the HW platforms for the network nodes were selected.

Figure 5: Use-case decomposed into modules

4.4.3 Build the prototype

The prototypes were built using an agile process and the development was carried out in several sprints. The result after each sprint was presented to an Anima employee. Each of the modules identified in section 4.4.2 were developed independently according to the descriptions presented in appendix E for Contiki and appendix F for mbed.
5 Results and Analysis

This section present both the data collected during the work of this thesis and also the analysis of the IoT OSs.

5.1 IoT OS Requirements

In collaboration with Anima, a list of important IoT OS characteristics was formed. This list is based on interviews with representatives from Anima, and defines the requirements for the evaluation of IoT OSs targeted for resource constrained battery driven devices.

1. Communication Protocols. Support for IPv6 (1.1), 6LoWPAN (1.2) and CoAP (1.3)

Using standardised communication protocols for IoT devices is essential for the growth of IoT. The low-level Internet protocols, typically residing in the network layer (Layer 3 according to the OSI model), such as IPv6 and 6LoWPAN serves as a good base for future IoT OSs to be built upon. However, IP protocols becomes a problem within battery powered IoT networks, this since the IP standard is not designed to be energy efficient. An IoT OS should support protocols based on the IEEE 802.15.4 radio technology that adapts IPv6 to the IoT for low-power and low data rate networks. Such protocols include 6LoWPAN and CoAP which both comply with the IPv6 standard.

2. Communication Radio. Support for IEEE 802.15.4 (2.1) and BLE (2.2)

Radio solutions for resource constrained IoT devices should focus on energy efficiency rather than on high data throughput. The IEEE 802.15.4 standard is designed for low-power, low data-rate wireless networks targeting the IoT domain. The introduction of Bluetooth Low Energy (BLE) and the ability to use IP over BLE is another advancement targeted for low-power wireless networks. Both of these radio solutions should be supported by an IoT OS.

3. Hardware Support. Support for at least one of the following HW platforms: Cortex-M (3.1), AVR (3.2), MSP (3.3)

Due to the large variety of possible use-cases within IoT, a large variety in HW platforms will need to coexist within the IoT space. Therefore, an IoT OS must be able to support several different HW architectures and platforms. Many HW platforms designed for IoT are built using ARM’s Cortex-M, Atmel AVR and Texas Instruments MSP series of processors, hence support for these MCU’s are desirable for an IoT OS. HW support is also affected by the design choices of the IoT OS, an OS with good hardware abstraction will require less effort to port to another HW platform.

4. Programming Model. Support for development in C (4.1) or C++ (4.2)

In order for an IoT OS to be developer friendly and ease rapid development it must allow development in already well-adopted languages. Additionally, since many devices targeted for the IoT are highly constrained, the supported languages must allow for memory efficient programming. Key languages to be supported by an IoT OS is therefore ANSI C and C++.

5. Memory Footprint. Support for heavily constrained devices within IETF specified class C1 (5.1) or C2 (5.2)

Many devices targeted for the IoT are expected to become smaller and cheaper rather than more powerful, hence these devices are expected to continuously have very limited resources. It is therefore important for an IoT OS to support these constrained devices, and in order to do so the memory footprint of the IoT OS must be kept small.

6. Energy Efficiency. Idle mode current usage less than 10 µA (6.1)

For IoT devices powered by batteries, energy efficiency becomes essential, and is something that should be taken into account when designing an IoT OS. By letting the OS handle the power management, applications can be developed with less focus on power management aspects. Although power consumption largely depends on the type of hardware used, it is desirable that the OS should provide functionality to keep the power consumption at a minimum. In order for a battery powered device to achieve an operation time of over one year in idle mode the device should use less than 10 µA.
5.2 Prototype Use-Case

The context of the prototype use-case is within home automation and aims to showcase some of the functionalities that may be required in order to design a "smart home" system. It is designed to evaluate the IoT OSs with regards to the requirements stated in section 5.1. The use-case is build around the nodes, these are equipped with sensors and they are controlling a lamp. The lamp can be turned on when a button is pressed on a trigger-device if the trigger device is within range of the lamp. To determine if the trigger device is within range of the lamp both devices would need BLE support. The trigger device could be a smartwatch or a smartphone. Additionally, if the trigger device is a smartphone it should also be able to request sensor data from the node.

5.3 Prototype Architecture

The prototype architecture, illustrated in figure 6, consists of several battery driven nodes running an IoT OS. The nodes are connected to a low-power wireless IP-based network. They connect to the Internet, and can be accessed with a web based client hosted on a web server. All nodes are equipped with sensors, a LED and BLE radio. The sensors read temperature and brightness data to showcase monitoring capabilities. The LED is used to represent a light bulb and the BLE radio is used for creating BLE beacons so that devices close to the node can detect it and then communicate with it via the Internet. In order to send or receive data all nodes host their own CoAP server. From the web based client the nodes’ sensor readings can be requested and displayed. The web based client also have buttons to post CoAP request to the nodes in order to toggle the LED's.

![Figure 6: Prototype architecture](image)
5.4 Contiki

This section presents information related to the data obtained for the IoT OS Contiki along with the analysis of that data.

Manufacturer data
According to the official Contiki documentation [30], Contiki is an open-source, lightweight IoT OS designed for resource constrained embedded systems. Contiki is released under a three-clause BSD style license [31]. A basic implementation of Contiki requires about 10 kB RAM and 30 kB ROM and is dependent on what OS modules are included in the build. Contiki’s open-source nature allows for open access to both documentation and source code, along with a large online community. Contiki offers both multithreading and optional preemptive multithreading and its programming model is based on protothreads, which are lightweight stack-less threads [32]. The combination of an event-driven kernel and preemptive multithreading allows for both resource efficiency as well as for the possibility to allow multiple threads. Contiki is designed to run on low-power battery driven HW platforms, where the system may need to run for years on batteries. This is achieved by the ContikiMAC protocol [33], which is a protocol that handles radio duty cycling and ensures that Contikis power consumption is held at minimum. Contiki also provides mechanisms for estimating a system’s power consumption and helping the developer to understand where power is consumed. Contiki offers support for a few standardized wireless communication protocol stacks, including TCP/IP, uIPv6 and the custom Rime stack. The uIPv6 stack, along with the RPL allows for mesh networking and 6LoWPAN communication. Contiki supports a range of different MCU and SoCs from different companies, including Texas Instruments (TI), ARM and Atmel. Among the supported HW platforms are TI MSP430, AVR STM32w, LPC2103, Microchip PIC32 and TI CC2650. Apart from traditional MCU support, Contiki is also able to be run on x86 based systems through simulation software.

Minimal Application - Experiences
Developing the minimal application for Contiki was done by following the guide as described in appendix A. It took less than two work days to set up the development environment and to run the minimal application. Finding and installing the compiler, cloning the Github repository along with installing the required dependencies took about two hours. Developing for Contiki differs from standard embedded programming since it does not start the execution of the application from a “main” method. Instead, the application code is placed in protothreads which are scheduled and later started. Since this development approach was new the development of the application was postponed one work day in order to learn the way Contiki applications are written and how the processes work. Compiling and flashing the application for Contiki took less than five minutes and was done using one single command line entry.

Interviews
Eistec AB, Joakim Nohlgård
Eistec AB is a company active in the IoT domain. Their main product is an embedded Internet system called Mulle. They have an ambition to use both RIOT and Contiki in their products. Eistec views Contiki as a mature IoT OS with a large user-base. However, most of its users are active within academia, where the focus is more on research than on actual product development. The largest benefit of Contiki is that it has come a long way in its development and Eistec is currently using Contiki in their products. However, they believe that Contiki has a major disadvantage when it comes to hardware abstraction; a driver written for a Contiki device will only work on that device. This since Contiki does not abstract any of the HW ports or buses. The company is involved in the development of Contiki but not to the same extent as they are for RIOT.

Thingsquare, Adam Dunkels
Thingsquare is a company developing IoT applications. One of the founders, Adam Dunkels, is also the creator of Contiki OS. Thingsquare is using Contiki in their products and over one hundred thousand HW chips developed by Thingsquare has been shipped to consumers according to Dunkels.
Yanzi Networks AB, Eva Fors

Yanzi networks AB is a company which develops IoT products targeted for "Smart homes" and "Smart offices". Their product suite consists of several different sensors and actuators, some of which run Contiki, in a low-powered IEEE 802.15.4 wireless network. Yanzi's view on Contiki is that they believe it to be a future-proof IoT OS due to the Contiki developers strong belief in RPL and IPv6 together with an actively working community. One of the main reasons why Yanzi uses Contiki in their products is due to the open-source nature of the OS and Yanzi is actively contributing to the open-source development.

Prototype - Development Experiences

The creation of the Contiki based prototype is described in appendix E and the test cases for the prototype are presented in appendix G. Data from the prototype running Contiki, shows that Contiki offers support for both CoAP and 6LoWPAN over IEEE 802.15.4. However, the BLE support is very limited since Contiki is missing a BLE stack implementation. Contiki offers a complete 6LoWPAN mesh-network solution, with Contiki running on nodes and a Contiki based application for the border router. During the creation of the Contiki prototype a major bug was identified in the Contiki source code, which made it impossible for the nodes to communicate with the border router. It took us five working days to identify a solution. A current usage analysis was conducted on the Contiki node, and this analysis showed that Contiki's current usage in idle mode is roughly 6 µA.

Evaluation

Based on the manufacturer data Contiki offers support for all the communications protocols specified in the requirement list. However, building the prototype for Contiki showed that the BLE support is partial at best. The lack of a BLE stack makes the use of BLE limited to a BLE advertisement routine. On the other hand, 6LoWPAN over IEEE 802.15.4 and CoAP support was extensive. Further indication of Contiki's communication protocol support was given by the fact that Yanzi Networks list Contiki's IPv6 and RPL as two of the main reasons for using Contiki in their products.

The manufacturer data further shows that Contiki meets the requirements regarding supported HW. All of the required HW platforms are supported. However, as pointed out by Joakim Nohlgård at Eistec Contiki has a major disadvantage regarding HW abstraction, which makes porting of Contiki to new HW platforms extensive.

During the creation of the minimal application for Contiki it became obvious that writing application for Contiki differs from standard embedded programming in C and or C++. The usage of processes and the absence of a main method requires additional time to learn. In addition to the difference in standard embedded programming, it was necessary to use to a specific version of the Contiki source code in order for the nodes to connect properly to the border router. Using the Contiki source code and the provided APIs provided easy access to the supported features and after learning the Contiki processes and using the right version writing application was done fast.

Contiki does offer support for IETF class C1 and C2 constrained devices. This was shown both by the supported platforms and was further confirmed since the prototype was able to be constructed on a highly constrained device residing in the C1 class. After building the prototype and conducting the current usage analysis it was shown that Contiki meets the requirements regarding Energy efficiency. The current usage in idle mode was measured to 6 µA.

The large user base of Contiki together with the good support for communication protocols and HW point towards Contiki being a future-proof IoT OS. The written interviews further strengthen this since three companies are already using Contiki in their products. All of which are also actively contributing to Contiki.
5.5 mbed

This section presents information related to the data obtained for the IoT OS mbed along with the analysis of that data.

**Manufacturer Data**

Mbed is a lightweight OS developed by ARM and is designed for resource constrained embedded HW platforms [34]. Developing applications for mbed is done by using the mbed SDK, an open-source SDK written in C and C++, released under the Apache licence 2.0. The provided mbed APIs enable re-usability of code since the applications are not hardware specific. Additionally, software development for mbed is OS agnostic due to the web based development tools. Offline software development is also supported through various tools and compilers [35]. Mbed’s network capabilities offers a wide range of protocol stacks, including well adopted and standardized protocols for both wired and wireless communication. The wireless protocols include BLE, Wi-Fi, 6LoWPAN, Thread, Zigbee IP, Zigbee NAN and Cellular, and for wired connections mbed supports Ethernet [36]. Mbed only supports ARM’s own Cortex-M architectures, which consist of processors optimized for low power consumption and low production costs [37]. The Cortex-M family of processors ranges from the highly constrained Cortex-M0 up to the more powerful Cortex-M7. Furthermore, ARM also provides a Hardware Development Kit (HDK) [38], enabling for custom Cortex-M based HW designs supported by mbed.

![mbed structure](image)

The architecture of mbed, seen in figure [7] is by default event driven and single-threaded. However, mbed offers optional support for implementation of real-time functionality. By utilizing this architecture mbed provides several abstraction layers for SW development. The two low-level layers allows mbed to be adopted to different HW platforms by providing the HW drivers and the CMSIS layer. The three communication layers allows for the use of standardized communication protocols to be implemented. Furthermore, the Device Management layer together with the C++ APIs allows for power management to reduce energy consumption.

**Minimal Application - Experiences**

Without any prior knowledge of mbed it took less than one work day to set up the development environment and run the minimal application according to the guide in appendix [B]. The main utility used for development for embed is Yotta, which required the installation of eight additional tools. The full installation of Yotta, including the dependencies took less than two hours to complete. Since Yotta was a new utility an additional four hours were required to learn how Yotta works and how to use it. Developing the application for mbed was done in C and required less than five minutes since mbed applications are developed like any regular embedded C program. Compiling and uploading applications for mbed was done in two steps. The first step took less than five minutes and was to compile the program using Yotta. The second step, uploading the application took less than one minute since we only needed to drag-and-drop the binary file to the JLINK USB mass storage of the target HW.
Prototype - Development Experiences

Mbed has extensive support for BLE with a full implementation of the BLE stack, but at the moment no support for IP over BLE. However, IP over BLE is listed on the roadmap for future implementation [39]. Mbed has SW support for 6LoWPAN communication over IEEE 802.15.4 radio. The 6LoWPAN solution requires a radio module which was not available at the time of this evaluation. Interfacing with GPIO pins were easy due to the well documented API's.

Evaluation

According to the official mbed documentation the support for communication protocols and communication radio is extensive. However, during the building process of the prototype it was elucidated that neither IP over BLE nor 6LoWPAN over IEEE 802.15.4 was fully supported, both of which were listed under the communication requirements in section 5.1. IP over BLE lacked SW support while 6LoWPAN did not have any available hardware for developers to purchase. Since neither IP over BLE nor 6LoWPAN over IEEE 802.15.4 could be implemented none of the application layer protocols such as CoAP was tested.

Mbed is limited to ARM's Cortex-M series of processor architectures, and since ARM is developing mbed it seems unlikely that other architectures will be supported. Even though the Cortex-M series of processors includes a range of different processors this limitation puts some restrictions as to which HW can be used when developing for mbed. Conversely, ARM does provide a mbed HDK which allows for easier development of custom Cortex-M based HW platforms supported by the OS.

Developing both the minimal application and the prototype for mbed showed that mbed has a thought-through programming model which makes writing applications for mbed fast and easy for developers accustomed to embedded C programming. The documentation for the different tools and the provided API's and libraries are good and the online community has a large user-base which further indicates that mbed indeed is a developer-friendly OS.

According to the Manufacturer data the memory footprint of mbed allows development of application targeted for IETF class C1 and C2 devices. However, since the connectivity for the prototypes could not be implemented we were unable to verify that the total memory footprint of the prototype application met the requirements. According to the data sheet of the nRF51 it meets the requirements regarding a current usage in idle mode less than 10 µA, but since the prototype for mbed was not finished we were unable to confirm how mbed affects the current usage of the system.
5.6 RIOT

This section presents information related to the data obtained for the IoT OS RIOT along with the analysis of that data.

**Manufacturer Data**

RIOT is a lightweight, open-source OS targeted for IoT devices. It offers low memory usage, real-time capabilities and support for both wired and wireless communication [40]. RIOT’s programming model has full support for C and C++ programming. Enabling applications to be written in a well-known programming language is a part of RIOT’s aim to be a developer friendly OS [9, 41].

![RIOT folder and design structure](image)

Figure 8: RIOT folder and design structure

Figure 8 shows the layer/folder structure of RIOT, and it illustrates RIOT’s hardware abstraction. Layer 1 to 3 are hardware independent. The `sys` folder contains APIs for memory-management, RIOT shells and more. The `sys/net` contains all the networking code for RIOT, and the `pkg` contains files that support third party libraries included in RIOT. Layer 3 contains the kernel and the hardware abstracted drivers. While layer 4 to 5 contains all the hardware specific code, where hardware peripheral drivers are found together with code for the supported CPUs and boards. RIOT has full support for multithreading and offers real time capabilities. Scheduling is handled using a fixed priority scheduling [42]. The power-management functions are handled with threads, when there is no more tasks for the OS to run, it automatically switches to the idle thread, which puts the board in deep-sleep mode [9]. RIOT has built-in network connectivity support, its network stack support IPv6, 6LoWPAN, UDP and CoAP. RIOT is developed by an open source community, and is not limited to any processor architectures. The memory footprint of RIOT is low, 1.5 kB ROM and 5 kB RAM, which enables it to run on small 8-bit processors. However, RIOT also supports 16-bit and 32-bit CPUs and new board support is constantly added by the community. It currently supports 37 different HW platforms including AVR, MSP and ARM Cortex-M MCU architectures. RIOT is under constant development and it is updated by the developing community several time a week [43].

**Minimal Application - Experiences**

It took less than one work day to set up a development environment for RIOT, which was done with no prior knowledge about the OS. Applications are written as standard C programs, since we have knowledge about development iC a simple RIOT application could be constructed within a couple of minutes. However, when creating the minimal application a serious bug was discovered in RIOT related to the serial port communication. This bug delayed the application development for two days, before the bug was fixed by the community. RIOT comes with configurations for flashing applications to supported platforms, this makes flashing very easy, which was achieved in less than five minutes. The built-in terminal that run on the development system was also up and running in less than five minutes. RIOT’s API’s are well documented. Appendix C contains a description on how to create a minimal application for RIOT. The documentation of supported platforms for RIOT varies, this can make it difficult to know if a HW-platform supports all the OS features.
Interviews

Eistec AB, Joakim Nohlgård

Eistec AB is a company active in the IoT domain. Their main product is an Embedded Internet System called Mulle. They have an ambition to use both RIOT and Contiki in their products. However, they have concluded that the current state of RIOT does not offer all features they need for their applications. The lack of support for radio duty cycling, MAC layer support and insufficient power saving functionalities makes it unsuitable for battery powered applications. However, the company is deeply involved in the development of RIOT and Joakim Nohlgård is also a maintainer in RIOT’s development programme. The company’s main reason to invest in the development of RIOT is that they believe that it shows potential as an IoT OS, mainly because of smart design choices, such as complete hardware abstraction. A driver for a device developed in RIOT is possible to use on multiple different devices, as long as the device has enough memory and all low level CPU interfaces, which is possible thanks to the abstraction of all HW ports and buses. Although Eistec’s ambition is to use RIOT they do not use it in any products today.

Evaluation

The manufacturer data shows that RIOT has support for all the communications protocols specified in the requirement list. This is also true for the HW support, where all the HW architectures in the requirements list are supported by the OS. However, the minimal application showed that not all the supported platforms are fully implemented by the OS, which can lead to confusion since this is not always specified in the OS documentation. As pointed out by Joakim Nohlgård RIOT has a well designed HW abstraction layer, which makes it easy for developers to port the OS to new HW platforms.

According to manufacturing data RIOT is a developer friendly OS, this was confirmed by setting up a minimal application which required less than one work day. But the minimal application also shows that there is a high potential for bugs within RIOT. In our very limited application we came across one major bug. A conclusion from this is that RIOT still is under development but that the programming model shows good potential.

RIOT’s memory footprint is within IETF class C1 and C2 and, out of the evaluated OSs it is the one with the smallest memory footprint according to manufacturer data. Although the manufacture data indicates that RIOT has power management functionality Joakim Nohlgård argues that these functionalities are insufficient for making RIOT a candidate for consumer products. Joakim Nohlgård believes that RIOT is a future-proof OS and the fact that Eistec invest a lot of time in developing the OS, strengthen this fact. Its manufacture data also shows that already include a lot of the features from the requirement list, but the fact that some important functionality is missing makes it not yet production ready. The absence of a radio duty cycle protocol currently limits RIOT’s ability to run on battery driven devices, since this leads to high energy consumption from the radio communication.
5.7 Zephyr

This section presents information related to the data obtained for the IoT OS Zephyr along with the analysis of that data.

**Manufacturer Data**

The Zephyr Project is an open-source project released under the Apache licence version 2.0, driven by the Linux foundation to develop a secure, small IoT OS targeted for constrained devices \[44, 45\]. The architectural model of Zephyr OS is a divided kernel model, Zephyr’s kernel is split into a nano kernel and a micro kernel. The nano kernel can be used separately or together with the micro kernel \[46\]. The nano kernel supports highly resource constrained devices, and offers limited multithreading capabilities. The micro kernel can be used on top of the nano kernel, which requires more memory but offers full support for multithreading. The micro kernel includes a scheduler that handles tasks similar to a standard RTOS. The micro kernel also supports dynamic memory allocation \[46\]. Only using the nano kernel Zephyr OS is able to run on devices with as little as 8 kB RAM, while using both kernels require an additional 50 kB. The supported hardware architectures of Zephyr OS includes ARM’s Cortex-M, Intel’s x86 and ARC. Currently six MCUs are fully implemented, including Arduino 101, Arduino Due and Minnowboard Max. Writing applications for Zephyr is done in C or Assembly and can be done using existing tools and compilers. The Zephyr API supports functions for connectivity and putting the device in idle mode when no more tasks are available \[44\]. Zephyr offers support for BLE, while other protocols such as WiFi, IEEE 802.15.4, 6LoWPAN, CoAP, IPv4, IPv6, and NFC are still under implementation.

**Evaluation**

Zephyr currently lacks several implementations when it comes to connectivity. The BLE stack seems to be the only stack fully implemented, which limits Zephyr regarding connectivity options. The memory footprint of Zephyr depends largely on the configuration, if only the nano-kernel is used Zephyr’s footprint allows it to be run on constrained devices. However, the supported HW platforms are rather limited, with only six MCU’s currently supported. The supported HW platforms consists of higher-end of the IEFT class C2 devices e.g. Arduino 101, up to high-end IoT devices e.g. Minnowboard Max. The lack of low-end HW support further limits Zephyr for low-end applications. Especially within battery powered use-cases since all of the supported boards have a deep sleep current usage above the requirement of 10 μA.

Zephyr offers extensive documentation which makes developing application easier and the programming model allows Zephyr to be considered developer friendly. However, since no connectivity solutions currently supported offer support for IP based networking and due to the scarce and high-end HW support Zephyr can not be considered a future-proof IoT OS.
5.8 Evaluation summary

This section contains a tabular summary of the IoT OS evaluation, with regards to both what types of data that was obtained and what requirements were met by the IoT OSs. Table 1 presents what data types that were collected for each OS.

<table>
<thead>
<tr>
<th>OS</th>
<th>Manufacturer Data</th>
<th>Minimal Application</th>
<th>Written Interviews</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contiki</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mbed</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RIOT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Zephyr</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2 presents the evaluation result in terms of what requirements each IoT OS met. The requirements that are met are represented by a check mark next to the requirement number. The full list of requirements are found in section 5.1.

<table>
<thead>
<tr>
<th>OS</th>
<th>Req.1</th>
<th>Req.2</th>
<th>Req.3</th>
<th>Req.4</th>
<th>Req.5</th>
<th>Req.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contiki</td>
<td>1.1 ✓</td>
<td>2.1 ✓</td>
<td>3.1 ✓</td>
<td>4.1 ✓</td>
<td>5.1 ✓</td>
<td>6.1 ✓</td>
</tr>
<tr>
<td></td>
<td>1.2 ✓</td>
<td>2.2 ✓</td>
<td>3.2 ✓</td>
<td>4.2</td>
<td>5.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3 ✓</td>
<td></td>
<td>3.3 ✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mbed</td>
<td>1.1 ✓</td>
<td>2.1 ✓</td>
<td>3.1 ✓</td>
<td>4.1 ✓</td>
<td>5.1 ✓</td>
<td>6.1</td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>2.2 ✓</td>
<td>3.2</td>
<td>4.2</td>
<td>5.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td></td>
<td>3.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RIOT</td>
<td>1.1 ✓</td>
<td>2.1 ✓</td>
<td>3.1 ✓</td>
<td>4.1 ✓</td>
<td>5.1 ✓</td>
<td>6.1</td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>2.2</td>
<td>3.2</td>
<td>4.2</td>
<td>5.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td></td>
<td>3.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zephyr</td>
<td>1.1</td>
<td>2.1</td>
<td>3.1</td>
<td>4.1</td>
<td>5.1</td>
<td>6.1</td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>2.2 ✓</td>
<td>3.2</td>
<td>4.2</td>
<td>5.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td></td>
<td>3.3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*It should be noted that Contiki only has partial support for BLE (2.2).*
6 Discussion

IoT is a fast moving area, which is also true for IoT OSs. The aforementioned should be taken into account when studying the results of this thesis. So, the results are a snapshot of the evaluated IoT OSs status in May 2016, and it is not possible to say for how long these results will be valid. New versions of the evaluated OSs are constantly released, in addition to this it is very likely that new IoT OSs are released in the future.

6.1 Method Discussion

Using a criteria based method provided a guideline on how to carry out the various tasks performed during the evaluation conducted in this thesis. The information gathered from interviews with Anima representatives was a valuable resource when identifying the IoT OS requirements. Those interviews helped us to narrow down the selection of IoT OSs within the scope of resource constraint devices.

By evaluating the IoT OSs on more than two types of data enabled the use of data triangulation, which provided a more detailed evaluation result than if only one data type would have been used. This is true for all OSs except for Zephyr which was evaluated on only one data source. However, a weakness of the method is that it does not evaluate all the IoT OSs on the same types of data. For both of the OSs which the prototypes were built on, shortcomings were found regarding the supported communication radios and protocols. In the case of mbed both IP over BLE and 6LoWPAN over IEEE 802.15.4 were found unsupported, and for Contiki BLE support was found minimal. If all OSs had been evaluated on all data types similar discoveries could have been made for all OSs. We argue that this elucidates the importance of a combination of a theoretical as well as an empirical evaluation for IoT OS. To ensure the validity of the prototypes they were developed according to an established scientific method, as explained in section 4.3 which strengthens the data gathered from the prototypes’ evaluations.

6.2 Validation of Requirements

The list of requirements, which the evaluation rests upon, was based on the inputs of representatives from Anima and the conducted literature study. This makes some of the listed requirements less generic and rather narrowed down towards IoT applications related to the area that Anima is active within, which is resource constrained wireless IoT devices. Therefore, these requirements are relevant to other companies active within the same area of IoT, something that was strengthened by the responses from the companies that answered the written interviews. Since the conclusions of the performed evaluation were based on these requirements the conclusions are also less generic but rather narrowed down to the usage of IoT application within battery powered use-cases for heavily constrained devices communicating over low power IP based networks.
6.3 Use-case Discussion

The use-case presented in section 5.2 provided a framework for the prototype construction. The purpose of the use-case was to enable prototype construction in order to gather experience from the building process of the prototypes. Given the scope of this thesis and the context of the use-case the experience from the use-case provided us with data regarding the following requirements:

1. Support for IPv6 (1.1), 6LoWPAN (1.2) and CoAP (1.3)
2. Support for IEEE 802.15.4 (2.1) and BLE (2.2)
3. Support for the Cortex-M (3.1) HW platform
4. Support for development in C (4.1) or C++ (4.2)
5. Support for heavily constrained devices within IETF specified class C1 (5.1) and C2 (5.2)
6. Idle mode current usage less than 10µA (6.1)

All of which are relevant for highly constrained battery driven devices connected to low-power wireless IP based networks.

6.4 Data Discussion

In the following four subsections discussions regarding the four data types collected during the evaluation are presented.

6.4.1 Manufacturer Data

All of the IoT OSs evaluated are under constant development, which means that support for new HW and communication protocols and radio are constantly added. Therefore, the manufacturer data can not be considered absolute, but rather provide a snapshot of how the OSs fared at the time of the evaluation. However, one of the requirements evaluated have a strong correlation to the future development of the IoT OS; The programming model plays a large role in how fast the development of the IoT OSs can progress.

6.4.2 Minimal Application

The results from the minimal application provided information on how much work was needed in order to set up the IoT OSs development environment. These results can be interesting for developers looking to use the IoT OS for product prototyping. The evaluation shows that it is likely that bugs affect the minimal application process. However, since the minimal application only covers a fraction of the functionalities that an OS has, it can not be used to determine how likely it is that bugs related to other OS functionalities exists.
6.4.3 Written Interviews

Eight companies using the evaluated IoT OSs were identified by the authors, out of them three replied to our questionnaire. We argue that the replies we obtained does provide valuable data with regards to resource constrained IoT devices. The reason why we find the responses valuable is because it represent the thoughts from developers experienced within the area. All of the developers who answered our questionnaire are working at companies active in the segment of IoT within resource constrained IoT devices. Each of these companies have developed and released products in this market segment, which makes the companies representative with regards to development experience for resource constrained and battery powered IoT devices.

We have identified three reasons why only eight companies were found: firstly, some companies might not want to openly promote what underlying technologies they are using in their products. Secondly, some companies might use proprietary SW solutions for their products. Finally, other IoT OSs than the ones included in the evaluation might be used in the industry.

The data we obtained from the written interviews has been a valuable addition to the evaluation of the OSs. By using the data from the written interviews we could determine that RIOT’s power management functionality was insufficient for resource constraint battery driven devices. Furthermore, it also provided a good indication that Contiki is a production ready IoT OS.

6.4.4 Prototype - Data Validity

Much of the data generated when building the prototypes is related to experience gained during the process of building the prototypes. We used three techniques to gather data from the prototypes:

1. The data related to the construction experience was gathered through taking notes about the construction process during the prototype creation.
2. Data about the prototype was also gathered through black box test cases. When deciding what to test the list of requirements was consulted, therefore only functionality important to our requirements list where tested.
3. Descriptive scenarios where used to create scenarios and observing the prototype’s behavior in these scenarios.

6.4.5 Prototype - Current Measurement

Due to the limitations of the Agilent 34410A, while using it with the BenchVue Digital Multimeter App, the sample time of the current measurements were limited to a minimum of 250 ms. This limitation can potentially have introduced inaccuracy to the recorded current measurements due to the fact that some fluctuations might not have been recorded. However, the measurements, seen in appendix G.6, does provide a good reference point of the system’s minimum, maximum and average current usage. The validity of the measurements were further strengthen since we were able to obtain similar data on different attempts. We also compared our result to official current draw data from the data sheet of the processor, and this comparison showed that our result was within range of the expected result from the data sheet.
6.5 Related Work

In section 3 we presented a summary of previous work, related to this thesis. Three papers were presented, two surveys and one evaluation. All of them base their evaluations/survey on theoretical data from different IoT OSs. We have chosen to evaluate on theoretical data, but also build a prototype to gain additional experience from the evaluated IoT OSs. In section 5.1 we presented a list of important IoT OS characteristics. The papers presented in related work also contains requirements for IoT OSs.

The gap analysis by Minerauda et al. [13] was based on six requirements out of which three partially corresponds to the requirements we identified. Support of heterogeneous devices and Developer support shows distinct similarities to the Hardware support and Programming model. While Data fusion & sharing shows partial similarities to Communication protocols. The remaining three requirements, Data ownership, Ecosystem formation and IoT marketplace, identified by the authors are less applicable for IoT OSs targeted for resource constrained and battery powered devices.

The paper by Hahm et al. [7] identified six important characteristics. Out of the six requirements Small Memory Footprint, Support for Heterogeneous Hardware, Energy Efficiency and Network Connectivity, were also identified as important characteristics during the evaluation of this thesis. However, Real-Time Capabilities and Security which both were identified by Hahm et al., were not deemed important for the evaluation conducted in this thesis. The main reason for this difference is that the paper by Hahm et al. emphasises more on time critical scenarios, which is of less importance in the context of highly constrained and battery powered IoT devices.

In the paper by Gaur, P and Tahiliani, M. [6] the authors list as set of IoT OS features they deem important. Programming model, Networking and Portability is listed as important features by the authors and strongly relates to our requirements Programming model, Communication protocols and Hardware Support. Gaur, P and Tahiliani, M also lists Architecture, Memory Management and Scheduling as important features, which are not listed in the requirement list formed during the evaluation of this thesis. We believe that one reason for this is that the paper by Gaur, P and Tahiliani, M is significantly focused on the low level design choices of the IoT OSs in contrast to our evaluation which emphasises on the important functionalities the IoT OSs should offer for resource constrained and battery powered IoT devices.

Due to the large variety of devices, applications and use-cases within the IoT area, the scope of the evaluations performed within IoT will have a large impact on the outcome of the results. However, several of the requirements deemed important in this evaluation and in the evaluations and surveys covered in related work, shows that even within such a large area as IoT there are common denominators deemed important. The support for: different types of hardware, standardised communication protocols along with a well designed and developer friendly programming model seems to be present in a majority of the research regarding SW development within the area of IoT.
7 Conclusions

The results of the IoT OS evaluation shows that there is no silver bullet for the IoT. None of the OSs evaluated can be considered to fully meet all of the requirements listed in section 5.1. This indicates that the IoT OSs are still immature and under development, which elucidates on the importance of research within the IoT OS domain.

7.1 Answering the Research Questions

In section 5.1 we present a list of six important IoT OS characteristics when choosing an OS for prototyping of resource constrained battery driven IoT devices. Four out of the six important characteristics within in the scope of this thesis, can be considered to be generic within the area of IoT.

Requirements 1 Communication Protocols and 2 Communication Radio are both essential within IoT, since the common denominator within IoT is the ability for devices to be connected to the Internet. Requirement 3 Hardware Support is also essential since the area of IoT is expected to include a plethora of different devices for a range of different use-cases. The last requirement deemed generic is requirement 4 Programming Model, which is important since it determines how developer friendly an IoT OS is. Requirements 5 Memory Footprint and 6 Energy Efficiency is less generic but rather narrowed down to devices and use-cases within the scope of this thesis. Both requirement 5 and 6 have a strong correlation with resource constrained and battery powered devices.

Three out of the four IoT OSs evaluated meets numerous of the requirements listed in section 5.1. However, only Contiki partially meets all requirements and is considered to offer the best solution for highly constrained devices connected to IP based low-power wireless networks. This is mainly due to its extensive support for low-power IP based connectivity solutions, but also because it is a more mature OS that is actually used within the industry. It does have a different programming model than the other evaluated OSs, which requires some time to get into. Overall Contiki is the most mature IoT OS out of the evaluated OSs in this thesis.

During the process of building a prototype based on an IoT OS a number of difficulties has occurred. Most of these difficulties relates to the fact that the area of IoT OSs is still immature and under constant development. For potential developers it can be difficult to understand how much of the functionality that is actually implemented by only reading the specifications, since the specifications often includes both the actual functionality implemented along with future features on the road map. Even solutions that are supported by the OS might not be possible to use, due to the lack of supported hardware or unclear directives on what hardware that is supported. By constructing a prototype based on an IoT OS, experience can be gained about the OS and regarding certain OS functionalities and how they can be implemented. An example of this is that Contiki lists that the CC2650 Sensortag supports BLE, but in reality the current BLE support is minimal.

The prototype construction also showed difficulties related to the lack of available HW. The mbed prototype in this thesis was not possible to construct due to an unavailable 6LoWPAN radio device. This leads to experience gained on what hardware that is available or fully supported by the OS.

Additionally, new features are constantly added to the OSs by the developing community and a problem with this is that the new features also introduces new bugs and those bugs may be detected by running the OS on a prototype. This phenomenon was observed on the Contiki prototype where the newest software version of the border router was not compatible with the latest Contiki OS version. Another example of this is when a minimal application was build using RIOT, this process uncovered a bug related to serial port communication. Potential IoT OSs candidates can be tested on a basic prototype to gain further knowledge about the current state of OSs and how well they are suited for a particular set of devices.
7.2 Further work

As previously stated, new versions of the IoT OSs in this evaluation are constantly released. Therefore a future evaluation like this one could provide a different result. To conduct a new evaluation after substantial updates have been released for the evaluated OSs could therefore provide additional insights on how the OSs have progressed in their development.

In this thesis we have evaluated four IoT OSs, which is just a small fraction of the available IoT OS on the market today, and new OSs are also constantly being released. To continue the work performed in this thesis a future evaluation could include different IoT OSs than the OSs evaluated in this thesis. Additionally, similar evaluations within different areas of IoT can provide useful information regarding other areas and other type of IoT devices.

Since the evaluation conducted in this thesis did not include any security aspects a future evaluation including such aspects would be useful in order to gain insights into how secure the different IoT OSs are.

7.3 Contributions of this thesis

The main contribution from this evaluation is the experiences gained from the evaluation process. By conducting an evaluation on IoT OSs, targeted for highly constrained wireless battery driven devices, we have shown both what difficulties that can occur during the process of building a prototype system and what experiences that can be gained from performing such an evaluation. Additionally we have, together with Anima, listed six requirements for IoT OSs targeted for resource constrained battery driven devices. An evaluation with regards to those requirements was conducted and the result of this evaluation was that as of May 2016 Contiki was the IoT OS that best meets the specified IoT OS requirements.

During the evaluation of the two prototypes a list of pros and cons was created for both Contiki and mbed. The pros and cons for the two OSs can be seen in table 3.

<table>
<thead>
<tr>
<th>OS</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contiki</td>
<td>Low power consumption</td>
<td>Minimal Bluetooth support</td>
</tr>
<tr>
<td></td>
<td>Low-power mesh IP networking</td>
<td>Different programming model</td>
</tr>
<tr>
<td></td>
<td>Hardware support</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Production ready</td>
<td></td>
</tr>
<tr>
<td>mbed</td>
<td>Developer friendly</td>
<td>No low-power IP networking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Highly dependant on ARM</td>
</tr>
</tbody>
</table>
References


A  Contiki Minimal Application

This appendix describes how to build, compile and upload a Contiki application to a Texas Instruments CC2650 Sensortag. This guide is written for development on Ubuntu 14.04.

Installing the Compiler
The CC2650 Sensortag is equipped with an ARM Cortex-M3 MCU and therefore a compatible arm-gcc-eabi compiler must be installed in order to compile the application. The gcc-arm-none-eabi-4.9-2015q3 have been tested and is confirmed to work with the CC2650 Sensortag. Installing the compatible compiler can be done by executing the following three steps;

1. Install 32-bit libraries (if developing on a 64-bit machine):
   
   ```bash
sudo apt-get -y install lib32z1 lib32ncurses5 lib32bz2-1.0
   ```

2. Download the Linux installation tarball "gcc-arm-none-eabi-4.9-2015q3-20150921-linux.tar.bz2" from the following web page:
   
   https://launchpad.net/gcc-arm-embedded/+download

3. Unpack and install the compiler in the `usr/local` directory of your computer:
   
   ```bash
cd /usr/local
sudo tar xjf ~/DownloadFolder/gcc-arm-none-eabi-4.9-2015q3-20150921-linux.tar.bz2
   ```

Cloning the Github repository
To build application for Contiki the repository containing the Contiki source code must first be cloned from Github:

   ```bash
git clone --recursive https://github.com/contiki-os/contiki.git
   ```

Creating the application
Contiki applications are written in C, however there is no "main" method that starts the application. Instead code is placed in processes which are then scheduled to run by the OS. These processes can run in parallel. Below is an example of an application that prints "hello world" through the UART.

```c
/*Minimal application hello world*/

PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);

PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_BEGIN();
    printf("Hello, world\n");
    PROCESS_END();
}
```
Compiling the application
A makefile is needed to build the application and in its simplest form it needs the following lines:

```makefile
CONTIKI_PROJECT = APPLICATION NAME
all: $(CONTIKI_PROJECT)
CONTIKI = ../PATH/TO/CONTIKI/ROOT
include $(CONTIKI)/Makefile.include
```

After the makefile is created the application can be compiled by using the make command from a terminal window:

```bash
cd /PATH/TO/APPLICATION_FOLDER
sudo make TARGET=srf06-cc26xx BOARD=sensortag/cc2650 FILE_NAME.bin CPU_FAMILY=cc26xx
```

Uploading the application
Compiling the application results in a .hex file which is the binary file containing the application. The last stage is to upload this file to the CC2650 Sensortag using the CC Debugger and CCS UniFlash.

Download and install CCS UniFlash from the following web site:


After finalising the installation connect the CC2650 Sensortag to the CC Debugger and connect the CC Debugger to the PC using a micro USB cable. Then open CCS UniFlash and go to File - New Configuration. Under Connection select Texas Instruments XDS110 USB Debug Probe, then select CC2650F128 under Board or Device. Then press OK, which will initiate connection with the CC2650 Sensortag. Wipe the flash memory by going to Flash Settings and press Erase Entire Flash. After the erase is completed it is time to upload the application by going to Programs and press Add. Locate the .hex file, select it and press OK, then Program.
B  mbed Minimal Application

This appendix describes how to build, compile and upload an mbed application to a Nordic nRF51 DK. This guide is written for development on Ubuntu 14.04.

Installing Yotta

Yotta is a command line based tool used for creating and managing applications for mbed. It is used to set up the application, download and import libraries as well as building the application using the target compiler. Installing Yotta was done according to the following two steps:

1. Install dependencies for Yotta:
   
   ```
   sudo apt-get update
   sudo apt-get install python-setuptools cmake build-essential ninja-build python-dev libffi-dev libssl-dev
   sudo easy_install pip
   ```

2. Use `pip` to install Yotta:

   ```
   sudo pip install yotta
   ```

Creating the application

Creating the application was done using Yotta in the following three steps:

1. Creating and entering the directory for the application

   ```
   mkdir hello-world
   cd hello-world
   ```

2. Initiate Yotta:

   ```
   yotta init
   ```

3. Enter descriptions for the application.

   When Yotta is initiated it will prompt the user to enter a set of descriptions about the application that is being initialised. Each statement ending with a ":" is a question which can be answered by typing the answer in the terminal and press "Enter". In this hello-world example not all descriptions were set. The ones set can be seen bellow from the following out-print:

   ```
   Enter the module name: hello-world
   Enter the initial version:
   Is this an executable (instead of a re-usable library module)? yes
   Short description: hello-world example
   Keywords:
   Author:
   Repository url (where people can submit bugfixes):
   Homepage:
   What is the license for this project (Apache-2.0, ISC, MIT etc.)?
   ```

When running `yotta init` and after setting the descriptions for the application a folder structure will be created in the current folder. The application code should be placed in the `Source` folder in a file named `main.cpp`. Writing applications for mbed is done like any standard embedded C programming. The application starts from an `app_start` method and the application code is written in .cpp and .h files. The following lines of code were written in order to construct a minimal application for mbed:

```
#include "mbed-drivers/mbed.h"
/*Minimal application hello world*/
Serial pc(USBTX, USBRX);
void app_start(int, char**)
{
    pc.printf("Hello World!\n");
}
```
Compiling the application
Yotta does not come with a compiler, so the build system needs the arm-gcc toolchain in order to compile
the application, which can be installed by running:

```
sudo apt-get install libc6-armel-cross libc6-dev-armel-cross
sudo apt-get install binutils-arm-linux-gnueabi
sudo apt-get install libncurses5-dev
```

The next step is to download the mbed module which is done by entering the following command in the
terminal:

```
yotta search module sdk
```

This command will produce a list of mbed versions available for download. Select the latest version and
enter the following command:

```
yotta install mbed-drivers
```

Now it is time to set the build target using the `target` command, which in this case it is the nRF5182-gcc:

```
yotta target nrf51dk-gcc
```

The application is now ready to be built, which is done using Yotta with the following command:

```
yotta build
```

When the build is finished a `.hex` file named "yourApplicationName-combined.hex" is produced in the
build folder.

Uploading the application
Connect the nRF5182 DK board to the PC with a USB cable. This will mount a USB mass storage with
a folder named `JLINK` on the build system. To upload the application simply drag and drop the binary
file into the `JLINK` folder.
C RIOT Minimal Application

This appendix describes how to build, compile and upload a RIOT application on a Zolertia z1 device. The development system in this guide is Ubuntu 14.04 and these instructions are based on the official RIOT documentation for setting up a build system and creating an application.

Compiler
To build a RIOT application the build system must have a toolchain for the targeted MCU, and the official RIOT wiki offers an advise to which toolchain that needs to be install for the different development boards. The Zolertia z1 is equipped with a Ti MSP 430 MCU, and therefore the MSP 430 gcc compiler must be installed in order to compile the application. Installing the MSP 430 gcc compiler was done by running the following terminal command:

```
sudo apt-get install gcc-msp430
```

Cloning RIOT
To build applications for RIOT the RIOT repository must be cloned form Github:

```
git clone https://github.com/RIOT-OS/RIOT.git
```

Creating the application
RIOT applications are written in standard C or C++. Writing applications for RIOT is pretty straight forward and any C embedded developer should feel right at home. All applications start in the main function, and can then be written like any other C embedded program. Since the HW is abstracted from the OS, the API’s should be consulted when developing the application. The application is the written in a standard .c file. Below is a trivial example of a hello world program. When writing more advanced applications RIOT’s well documented API provides good support.

```c
/*Minimal application hello world*/
int main(void)
{
    printf("hello world \n");
    return 0;
}
```

Compiling the application
For RIOT to compile your application a makefile needs to be created. This makefile should point out the RIOT main directory on your the development system and also include what modules your application is using. A very simple template makefile is presented below.

```
APPLICATION = //This is your application name
BOARD= //This is your board which your application should run on.
RIOTBASE ?= //RIOT directory on your system
USEMODULE == //Adds modules to your application.
include (RIOTBASE)/Makefile.include //Required at the end of your makefile.
```

There are many more RIOT related options and commands available for use in the makefile, the ones presented here are just the minimum makefile needed to build the application. With the application and makefile in place the application is ready to be compiled. Compiling the application is done by executing the `make` command.

1http://www.codeproject.com/Articles/840499/RIOT-Tutorial
2http://riot-os.org/api/
Uploading the application
All RIOT supported board comes with a solution for flashing binaries to the board. Flashing is done by
the make flash command. However, different board uses different flashing techniques and some of them
might need the user to install additional software, or perform other tasks to make them work. For the z1
board the TI goodfleet flashing tool is used, and this tool needs to be added to the path to work.

$PATH/path/to/RIOT/dist/tools/goodfet/goodfet.bsl /usr/local/bin

RIOT contains a built-in terminal that can be used to receive prints from the board, this terminal can
be executed by the make term command and will automatically set up a serial connection to the board.
Some boards also offer gdb debugging support, in those cases the debug server is included in RIOT.
D Written Interviews

This appendix presents the questions asked in the questionnaire sent out to developers working at companies active within the IoT field. In each of the questions "OS XX" is replaced with the name of the corresponding OS which the company is using.

- What OS characteristics were important when deciding to use OS XX in your products?
- How future safe do you consider OS XX to be (active open-source community, well developed, good design choices, etc.)?
- Have you considered to, or are you using any other IoT OS than OS XX in your current products?
- If yes, do you have any plans to switch from OS XX, or are you planning to use more than one IoT OS?
- Is OS XX a part of your final products or is it only used during the development of your products?
- Are you actively contributing to the further development of OS XX?
E Prototype Contiki

This appendix explains the building process of the prototype for Contiki. Building the prototype was done by following the modular design presented in figure 5 in section 4.4.1. This made it possible to develop each of the modules separately. Developing the modules for Contiki was done by using the development environment according to appendix A.

E.1 Connectivity Module

For the Contiki prototype 6LoWPAN over IEEE 802.15.4 was chosen as the low-power wireless network. Since no smartphones or computers have built-in support for 6LoWPAN over IEEE 802.15.4 a border router was needed in order for the nodes to communicate outside of the wireless network. After selecting the communication solution for the prototype the next step was to select the HW for the nodes.

Nodes

For the nodes it was essential to have a small HW platform that was able to run on batteries while still offer support for 6LoWPAN over IEEE 802.15.4. The TI CC2650 Sensortag was deemed a good fit due to its small form factor, energy efficient MCU, its built in sensors along with the dual radio support for IEEE 802.15.4 and Bluetooth. In order for the nodes to be able to receive requests and to send sensor data each of the nodes were developed to host their own CoAP server. This allowed the nodes to receive POST and GET requests. The Post requests were used to allow the nodes LED’s to be toggled and the GET requests were used to allow the nodes sensor data to be read remotely.

Border Router

For the border router it was important to find a more powerful HW platform that could handle the routing between the 6LoWPAN network and a standard WiFi network. The border router consisted of a TI CC2531 USB dongle attached to a Raspberry Pi running Raspbian together with a border router application called 6LBR. In the newest version of Contiki, available at the time of the prototype construction, there was a bug that prevented the nodes to register and connect to the border router. In order to get this to work a previous version of the Contiki source code and the border router software had to be used. We have confirmed that the following versions of Contiki and the 6LBR firmware works:

Border router software: Cetic 6LBR, version 1.3
Contiki OS: git branch 7dc8ace

Since the CC2531 USB dongle comes pre-installed with a packet sniffer application the first step was to download and install a the Serial Line IP (SLIP) radio application to CC2531. In order to program the CC2531 the CC-Debugger was used. Installing the SLIP radio application on the CC2531 was done in the three following steps:

1. Download and install Smart RF Programmer from:
   http://www.ti.com/tool/FLASH-PROGRAMMER

2. Download and unzip the SLIP radio application binary file from:
   http://processors.wiki.ti.com/images/1/10/Cc2531-slip-radio_contikimac.zip

3. Flash the .hex file using Smart RF Programmer.
   Under the tab Flash image locate the .hex file unzipped in the previous stage. Then make sure that System-on-Chip and Erase, program and verify is selected before pressing Perform actions.

After installing the SLIP radio application the next step was to install the border router application 6LBR to the Raspberry Pi. On the Raspberry Pi we downloaded and installed 6LBR v.1.3 from the following web site:

https://raw.githubusercontent.com/cetic/6lbr/releases/rpi/cetic-6lbr_1.3_armhf.deb
After installing 6LBR a few configurations were needed. In order to set these configurations the 6lbr.conf file was created in the /etc/6lbr directory. This was done in the following two steps:

1. Created the 6lbr.conf file using nano:

   cd /etc/6lbr
   sudo nano 6lbr.conf

2. Entered the following configuration lines into the newly created file:

   MODE=ROUTER
   RAW_ETH=1
   BRIDGE=0
   DEV_BRIDGE=br0
   DEV_TAP=tap0
   DEV_ETH=eth0
   RAW_ETH_FCS=0
   DEV_RADIO=/dev/ttyACM0 #CC2531 will enumerate like ACM0
   BAUDRATE=115200
   LOG_LEVEL=3

Once the 6lbr.conf file had been edited and saved the configuration was completed and the border router application was ready to be started, using the following command:

   sudo service 6lbr start

To confirm that the configuration was successful and that 6LBR had started the nvm_tool was used to read the nvm.dat file:

   /usr/lib/6lbr/bin/nvm_tool --print /etc/6lbr/nvm.dat

A successful configuration and start of 6LBR resulted in the following output:

```
Reading nvm file '/etc/6lbr/nvm.dat'
Channel : 26  // channel of the 6LoWPAN network

WSN network prefix : aaaa::
WSN network prefix length : 64
WSN IP address : aaaa::100
WSN accept RA : True
WSN IP address autoconf : True

Eth network prefix : bbbb::
Eth network prefix length : 64
Eth IP address : bbbb::100  // IP-address used to access the border router
Eth default router : ::
Eth IP address autoconf : False

Local address rewrite : True
Smart Multi BR : False

RA daemon : True
RA router lifetime : 0
RA maximum interval : 600
RA minimum interval : 200
RA minimum delay : 3
RA PIO enabled : True
RA prefix valid lifetime : 86400
RA prefix preferred lifetime : 14400
RA RIO enabled : True
RA RIO lifetime : 1800

RPL instance ID : 30
RPL Preference : 0
RPL version ID : 59
```
Web server and client website

In order to communicate with the nodes from a smartphone or computer a server was written in Node.js using the Node.js modules Express and node-coap. The server hosted a website written in HTML which allowed smartphones and computers to browse the website using any modern web browser. This website provided a GUI for interacting with the nodes on the 6LoWPAN network. Several buttons used for toggling the lights on the nodes were displayed on the website, along with text areas for the sensor data.

In order for the the machine hosting the server to accepts routing advertisements and routing advertisement options for IPv6 routing the following two commands were run:

```bash
sudo sysctl -w net.ipv6.conf.eth0.accept_ra=1
sudo sysctl -w net.ipv6.conf.eth0.accept_ra_rt_info_max_plen=64
```

E.2 Proximity Module

BLE was chosen to handle the proximity detection of the Contiki prototype. The objective was to make the nodes act as BLE beacons which sends out advertising packets in fixed time intervals. Since the TI CC2650 supports BLE no additional hardware was required. However, when trying to implement the BLE beacon functionality is was obvious that Contiki lacks a BLE stack. The only BLE functionality provided is a custom made beacon that is able to broadcast only its name. Since our objective was to use and evaluate existing features rather than implementing features to the OS the BLE beacon could not be completed for the Contiki prototype.

E.3 Power Consumption Module

The application for the Contiki prototype is written so that the device will enter idle mode when there are no tasks to execute. To further keep the power consumption to a minimum the ContikiMAC radio duty cycling was used.

E.4 Light Control Module

In order to control the LED connected to the node a module for this was constructed. This module consisted of mapping the GPIO connected to the LED with an incoming event triggered by a CoAP POST request.
F Prototype mbed

Building the embed prototype was done by dividing the work into modules according to figure 5 in section 4.4.1. The modules were then created separately. The hardware platform chosen for the prototype’s nodes was the NRF5182 DK which offers built-in BLE radio support. In the minimal application for mbed we constructed a development environment under Ubuntu 14.04. This environment was also used for the prototype creation.

F.1 Connectivity

We chose BLE as the connectivity solution for the mbed prototype, this decision was made to test mbed’s support for IP over BLE. However, during the implementation of the connectivity solution it became apparent that IP over BLE was not yet supported by mbed. Focus was then shifted towards a 6LoWPAN connectivity solution. Although this solution had better software support its was not possible to set up due to the lack of HW support. The only radio HW supported by mbed’s 6LoWPAN implementation was an IEEE 802.15.4 shield, which was not yet on the market when this IoT evaluation was conducted. This meant that none of the connectivity solutions, listed in the requirements, was possible to implement on our prototype.

F.2 Proximity

BLE was used for proximity detection. This solution worked well with mbed because of its extensive BLE support, with a fully implemented BLE stack and API. The mbed BLE API’s was used to create an proximity module. This was done by having a nRF51 board act as a beacon that advertise by sending a BLE advertising package within fixed interval. The package contains the name of the beacon device, and a specific UUID that identifies the device. The mbed BLE example application was used as a reference when creating this module.

F.3 Power Consumption

Since no IP based communication were successfully implemented there was no value in conducting power measurements for the mbed prototype.

F.4 Control light

The light control module was constructed using mbed’s LED service API. Which provides an interface to the LED’s in the HW platform.

G Contiki Test Cases

This appendix presents the test cases performed for Contiki. A summary and the results of the performed test cases is shown in Table 4.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT 1</td>
<td>Node router connection</td>
<td>Pass</td>
</tr>
<tr>
<td>CT 2</td>
<td>Ping node over IPv6</td>
<td>Pass</td>
</tr>
<tr>
<td>CT 3</td>
<td>CoAP POST request</td>
<td>Pass</td>
</tr>
<tr>
<td>CT 4</td>
<td>CoAP GET request</td>
<td>Pass</td>
</tr>
<tr>
<td>CT 5</td>
<td>BLE advertisement</td>
<td>Fail</td>
</tr>
<tr>
<td>CT 6</td>
<td>Current measurement</td>
<td>Pass</td>
</tr>
</tbody>
</table>

G.1 Node router connection

In this test case a node tries to connect to the border router. The connection process can be observed by logging in on the border router and via a GUI control if nodes are connected to the router. When the node is powered on it is expected to connect and register with the router. Figure 9 shows the GUI when no nodes are connected to the router. In figure 10 the node has been powered on and has successfully registered with the router.

Figure 9: No nodes connected to the border router

![Figure 9: No nodes connected to the border router](image)

Figure 10: One node connected to the border router

![Figure 10: One node connected to the border router](image)
G.2 Ping node over IPv6

This test case was done in order to make sure that the node was reachable from a computer by sending a ping request to the node from an Ubuntu terminal. To ping the node the `ping6` command followed by the node’s IPv6 address was used. This command pings the node by sending packages and waits for a response from the node. The expected results is that the node responds to the pings from the computer. The result of the ping6 command can be seen in figure 11 which shows 11 successful pings to the node.

![Ping requests to node](image)

Figure 11: Ping requests to node
G.3 CoAP POST request

In this test case a CoAP server is running on the node. A client written in Node.js was used to post data to a node with a CoAP POST request. The request contains data which tells the node to turn on the LED attached to it. Expected results is that the node responds to the request which confirms that is was successful. A successful example request can be seen in the following log output from the server:

------------------ REQUEST: ------------------------
Url {
  protocol: 'coap:',
  slashes: true,
  auth: null,
  host: '[aaaa::c30c:0:0:e7b]:5683',
  port: '5683',
  hostname: 'aaaa::c30c:0:0:e7b',
  hash: null,
  search: null,
  query: null,
  pathname: '/lon',
  path: '/lon',
  href: 'coap://[aaaa::c30c:0:0:e7b]:5683/lon',
  method: 'POST'}
------------------ RESPONSE: ------------------------
IncomingMessage {
  _readableState: 
    ReadableState { },
  readable: true,
  domain: null,
  _events: {},
  _eventsCount: 0,
  _maxListeners: undefined,
  payload: <Buffer >,
  options: [],
  code: '2.05',
  method: undefined,
  headers: {},
  url: '/',
  rsinfo: {
    address: 'aaaa::c30c:0:0:e7b',
    family: 'IPv6',
    port: 5683,
    size: 8 },
  outSocket: { address: '::', family: 'IPv6', port: 58257 },
  _packet: {
    code: '2.05',
    confirmable: false,
    reset: false,
    ack: true,
    messageId: 21269,
    token: <Buffer 30 b8 e9 eb>,
    options: [],
    payload: <Buffer > },
  _payloadIndex: 0 }
G.4 CoAP GET request

In this test case a CoAP server is running on the node. A client written in Node.js was used to request data from the node with a CoAP GET request. In this test case the client request sensor data from the node. The expected result is that the node responds to the request and that the response contains data from the sensor. A successful example request can is shown in the following log output from the server:

--- REQUEST: ---

Url {
  protocol: 'coap:',
  slashes: true,
  auth: null,
  host: '[aaaa::212:4b00:7a8:6201]:5683',
  port: '5683',
  hostname: 'aaaa::212:4b00:7a8:6201',
  hash: null,
  search: null,
  query: null,
  pathname: '/sen/tmp/obj',
  path: '/sen/tmp/obj',
  href: 'coap://[aaaa::212:4b00:7a8:6201]:5683/sen/tmp/obj',
  method: 'GET'
}

--- RESPONSE: ---

IncomingMessage {
  _readableState: {
    readable: true,
    domain: null,
    _events: {},
    _eventsCount: 0,
    _maxListeners: undefined,
    payload: <Buffer 31 38 2e 38 31 32>,
    options: [ { name: 'Content-Format', value: 'text/plain' } ],
    code: '2.05',
    method: undefined,
    headers: { 'Content-Format': 'text/plain', 'Content-Type': 'text/plain' },
    url: '/',
    rsinfo: {
      address: 'aaaa::212:4b00:7a8:6201',
      family: 'IPv6',
      port: 5683,
      size: 16,
    },
    outSocket: { address: '::', family: 'IPv6', port: 35287 },
  _packet: {
    code: '2.05',
    confirmable: false,
    reset: false,
    ack: true,
    messageId: 38691,
    token: <Buffer 85 c2 6a 36>,
    options: [ {Object} ],
    payload: <Buffer 31 38 2e 38 31 32> },
  _payloadIndex: 0
}
G.5 BLE advertisement

In this test case the BLE advertising is tested for Contiki. This was done by initiating BLE advertising on the node. Reading advertising packages was done with and iOS application from Nordic semiconductors. The application scans for nearby BLE beacons and reads the data from the available advertising packages. Expected results is that the application from Nordic semiconductors is able to read the name and UUID of the node and that the application can connect to the beacon. The data advertised from the Contiki beacon is shown in figure 12 which shows that only the name of the beacon is advertised and that the app is unable to establish a connection to the beacon.

Figure 12: BLE beacon Contiki
G.6 Contiki current measurement

In this test case the current usage of a node running a Contiki application was analysed. Figure 13 presents a block diagram over the measurement setup. A PeakTech 6080 powerbox was used as power supply, it supplied a voltage of 3.3 V and a maximal current of 3 A. To measure the current usage from the node an Agilent 34410A DMM with the application BenchVue Digital Multimeter Pro App was used to record the current usage over time. The node was running the Contiki prototype application described in Appendix E. The sample interval used was the minimum available for the DMM instrument (250 ms) and the current usage was recorded over a number samples when the node was running a Contiki application. The expected result of the test, is that the idle mode current draw of the a Contiki node, is less than 10 µA. The current usage can be seen in figure 14 and in figure 15.

Figure 13: Current measurement block diagram

Figure 14 shows how Contiki is switching between idle mode and active mode, and it is recorded a node executing Contiki. While conducting this measurement two CoAP requests was made to the node, this can be seen in the figure 14 where the current usage increases while Contiki is in active mode. To measure the current usage a more detailed plot over a smaller area is presented in figure 15. The objective of this test was to determine the maximum minimum and average current usage, which is presented in table 5. The average current is calculated by taking the sum of all the sample values and dividing it by the number of samples. To verify the measurement a comparison with the data sheet of the CC2650 MCU was made. According to the data sheet the CC2650 draw 1 µA in idle mode, and 6 mA when transmitting data. From figure 14 it can be seen that the highest peak is at approximately 6 mA, it can also be seen in table 5 that the idle mode current usage is around 6 µA.

Table 5: Contiki current statistics

<table>
<thead>
<tr>
<th>Minimum current</th>
<th>Maximum current</th>
<th>Average current</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 µA</td>
<td>171 µA</td>
<td>32 µA</td>
</tr>
</tbody>
</table>
Figure 14: Contiki Sensortag current when requesting two CoAP get-request

Figure 15: Contiki Sensortag current usage in idle mode
H mbed Test Cases

This appendix presents the test cases performed for mbed. Since the mbed prototype was not finished not all test cases were performed. A summary and the results of the performed test cases is shown in table 6.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT 1</td>
<td>Node router connection</td>
<td>Not performed</td>
</tr>
<tr>
<td>MT 2</td>
<td>Ping node over IPv6</td>
<td>Not performed</td>
</tr>
<tr>
<td>MT 3</td>
<td>CoAP POST request</td>
<td>Not performed</td>
</tr>
<tr>
<td>MT 4</td>
<td>CoAP GET request</td>
<td>Not performed</td>
</tr>
<tr>
<td>MT 5</td>
<td>BLE advertisement</td>
<td>Pass</td>
</tr>
<tr>
<td>MT 6</td>
<td>Current measurement</td>
<td>Not performed</td>
</tr>
</tbody>
</table>

H.1 BLE advertisement

In this test case the BLE advertising is tested for mbed. This was done by initiating BLE advertising on the node. Reading advertising packages was done with and iOS application from Nordic semiconductors. The application scans for nearby BLE beacons and reads the data from the available advertising packages. Expected results is that the application from Nordic semiconductors is able to read the name and UUID of the node and that the application can connect to the beacon. The data advertised from the mbed beacon is shown in figure 16, which shows that both the name, along with the UUID is broadcasted. Figure 16 also shows that the application is able to establish an connection with the beacon.

![BLE beacon mbed](image)

Figure 16: BLE beacon mbed