Delft University of Technology
Master of Science Thesis in Embedded Systems

Battery-Free Operation in Existing Building Automation Networking Protocol

Jeffrey Quinten Bouman

Embedded Systems

TU Delft Delft University of Technology

Battery-Free Operation in Existing Building Automation Networking Protocol

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Jeffrey Quinten Bouman

15-09-2023

Author
Jeffrey Quinten Bouman
Title
Battery-Free Operation in Existing Building Automation Networking Protocol
MSc Presentation Date
20-09-2023

Graduation Committee

dr. Chang Gao	Delft University of Technology
dr. Przemysław Pawełczak	Delft University of Technology
dr. Jasper de Winkel	Delft University of Technology

Abstract

The averse reaction of lithium batteries inspired researchers to look for alternatives for energy storage in electronics. Battery-free or intermittent devices introduce a new paradigm with their own specific problems about power, state tracking, timing and communication. We implemented the battery-free paradigm in the Thread protocol, utilizing the specific characteristics of End Devices in the mesh network to reduce power consumption. The intermittent implementation is fulfilled without changing the behaviour of the protocol, making the device fully compatible without other devices being aware of the intermittent operation. During tests four implementations are tested, completely naive, semi naive with simple power improvements, Sleepy End Device operating intermittently and a Synchronised Sleepy End Device operating intermittently. The results of the power measurements are used to determine the feasability of four use cases. The intermittently operating Sleepy End Device is out performing the Naive and Semi Naive implementation.

Preface

The inspiration for this topic was sparked during a presentation in one of the classes, where research was shown on FreeBie. Inspiring me to apply the paradigm to one of my passions, building automation.

I am thankful to dr. Chang Gao for providing their knowledge and expertise. I am also grateful to dr. Przemysław Pawełczak and dr. Jasper de Winkel for their guidence during the making of this thesis. Lastly, I would be remiss in not mentioning friends and family for their support during the making of this thesis.

Jeffrey Quinten Bouman

Delft, The Netherlands
21st September 2023

Contents

Chapter 1

Introduction

The intermittent computing paradigm tries to step away from batteries by utilizing different energy storage options, often capacitors or super capacitors. The trend to move away from batteries has multiple different reasons. The first issue with batteries, especially the Lithium-ion (Li-ion) type, is the averse reaction to mechanical and temperature stress. When Li-ion batteries are subjected to these types of strain, the battery might burst. Secondly, batteries are dangerous when they start leaking or get damaged. Especially when the devices utilizing such a battery is used in a hard to reach place, like a satellite, in concrete or inside a human body [28]. Lastly, the replacement of a battery in such hard to reach places can be costly. At the moment a battery meets the end of its life, a replacement is necessary, which is not always feasible. In these type of scenarios an intermittent system with a capacitor which does not have the same drawbacks as batteries, is an excellent solution.

1.1 Intermittent Origins

The roots of intermittent programming and systems come from the RFID world. Where systems are designed to work with energy gathered from a wirelessly transmitting source. This energy source is not always present, so the system needs to be designed to directly start operating on the task at hand. However, these RFID based systems do not need to gather information while disconnected from their power source. Also is their wireless connectivity based on the backscatter of the source device. From there, research slowly transferred into other directions with systems utilizing different energy sources, like photo-voltaic cells and power generating buttons like the ones found in the ENGAGE [6] project.

1.2 Challenges

Designing intermittently operating devices and writing code for such devices is greatly different from designing a system which can continuously draw energy from a battery. These devices are only turned on for short periods of time with relatively long delays in between, see Figure 1.1 as an example. Therefore completely different pitfalls appear. Some of these challenges consist of, but are not limited to; time keeping, memory consistency, energy awareness, scheduling

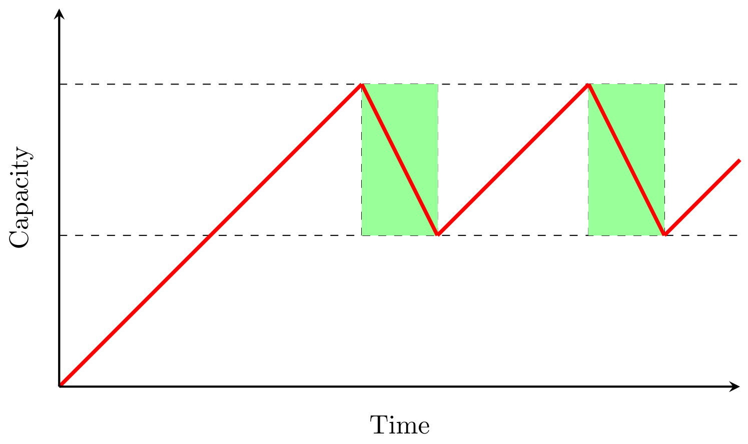

Figure 1.1: **Intermittent operation example with the red line indicating the capacity of the device and the green rectangles the on time of the device. At the capacity of the upper dashed line, the device turns on and at the lower dashed line the device turns off again.**

and wireless communication. For most of these challenges are multiple solutions possible, some having both a software as well as a hardware solution. When combining the solutions of these challenges, they might interfere with each other.

1.3 Goal

How can the characteristics of the Thread protocol be utilized to create battery-free operating devices?

To start of, the current state of battery-free device technology described in Chapter 2.

Secondly, The communication protocol chosen for this research as well as three use cases in which the protocol is used, are written down in Chapter 3. In order to utilize the protocol and take advantage of its possibilities, a more detailed insight of our approach is given in Chapter 4.

Next we gathered power consumption data on the devices operating in different network configurations. The power consumption per configuration is compared against each other. Information about the test configuration, the results of these experiments and the impact on our use cases are given in Chapter 5.

Lastly, the thesis is concluded and discussed in Chapter 6.

Chapter 2

Related Work

Intermittent operation is a niche subject with specific problems. These different challenges can be roughly divided into two subjects, the hardware and software side. Below we discuss the different challenges within these subjects.

2.1 Hardware

A convergence to an all encompassing hardware platform will greatly increase the ability for researchers to compare different software implementations, without having to account for differences in hardware designs. We need a platform which has the capabilities to receive power from multiple different sources and can communicate over a variety of wireless communication protocols.

2.1.1 Platforms

In the past, a few hardware platforms were often used to compare different software implementations with each other, for example Moo [42] and WISP [34]. However these platform mainly focus on the RFID type of devices. RFID based platforms do only react to incoming messages by deflecting the power in different ways or depend on power from an induction source. The closest solution for new, non RFID, systems found in academia is, for example the Flicker [19] system, which makes it possible to interchange different sensors on a batteryless stack. However, due to their custom design and inability to change out storage types, make the demand for interchangeable hardware is hard to fulfill.

2.1.2 Energy Storage

Intermittent systems still need to store power somewhere, in an overwhelmingly number of cases this energy is stored in capacitors. Even though these type of systems are often referred to as batteryless systems, some devices can technically still operate from energy stored in a battery. The main reasons to stop using batteries in systems are already mentioned earlier. However one could also reason for the opposite. Systems designed to use sensors (e.g temperature) and oscillators which drift over time might be a bottleneck, preventing perpetual operation. Therefore designing a system containing batteries with an expected lifetime longer then other components in the same system, might relieve a programmer of the burden for writing hard to write intermittent code. Especially newer battery technologies as shown in [22], like Lithium Titanate and Lithium Iron Phosphate, which remove the mechanical and thermal constraints of existing Lithium battery technologies.

Although Batteries might still be relevant in some cases, most research is done with capacitor based systems, these systems can divided into two main paradigms; distributed and centralized. In a distributed energy environment, each sub device in the system has its own energy storage. This approach gives more flexibility for subsystems to operate independently, with the drawback of introducing a synchronization problem between the different subsystems, GRANT [41] is a possible solution to this problem. In centralized designs, all subsystems drain consume energy from a single point, which can still be turned of for specific parts of the device. For example a real-time clock consumes so little power, it can be kept on even when the MCU is turned off, as seen in the design of FreeBie [7]. Research towards micro management of the energy capacity is also done with Capybara [4].

2.1.3 Harvesting Energy

Intermittent systems gain their energy from different types of sources which do not provide energy at a fixed voltage or current. In order to store the energy correctly, special harvester hardware is needed to charge a capacitor or other storage component. These harvesters can gain the energy from different sources which include, but are not limited to; photo voltaic cells, piezo crystals, generators and radio frequencies. All with their own advantages and drawbacks. An overview of different currently used harvesters and energy sources is made in [23].

2.1.4 Keeping Track of Time

Time keeping is, as mentioned before, one of the main problems to tackle for intermittent systems. Devices need to keep track of time while turned off, as well as being able to turn on at specific intervals for correct measurements and communication. For now two paradigms are possible.

The first is to use a capacitor to relate time. Capacitors can be configured to have a static discharge rate as achieved by BOTOKS [5]. Combined with the direct relation between stored energy and voltage, make capacitors a candidate for determining elapsed time when the MCU is not running. The advantage of this technique is the ability to have different granularities compared to the next option. However, the capacitors need to be recharged and have a fixed maximum time which they can measure before running out of energy.

The second option is to use a Real Time Clock (RTC). This RTC consumes significantly less power compared to the deepest sleeping states of microcontrollers. Such a solution is implemented in the mote used by FreeBie [7]. Therefore, the RTC can keep operating while the MCU is turned off completely. The advantage of an RTC over a capacitor is the ability to set a timer. The timer can make sure the MCU can be turned on at specific irregular intervals (if enough power is stored) and be woken up at these specific timings. Also is the accuracy of the RTC better compared to a capacitor, which timing determination is correlated to the accuracy of the used Analog to Digital Converter (ADC). The major reason to refrain from using an RTC is the operating current. Even though the current is low. The system still consumes power.

2.1.5 Non-Volatile Memory

During any point of operation an intermittently operating device can run out of energy to operate. In order to prevent total loss of all data, intermittent systems are designed to continue operating from a previous point in time. Information about where this point (stack pointer and program pointer) as well as other system memory (registers and (S)RAM) needs to be stored in a medium which does not lose it's content when powered down. For this non-volatile memory type two different approaches are used. Either the information is stored in long term storage like flash or EEPROM, which take a long time to store its information and takes up a lot of energy compared to internal SRAM [9]. Or the information is stored in a version of RAM which does not lose its content when powered down, for example FRAM. This type of RAM operates a lot quicker compared to different ROM technologies [9] and consumes a lot less power. This is not a silver bullet though, the writing and reading speeds are slower compared to SRAM and is more expensive to purchase per byte. Deciding when to write data to Non-Volatile Memory (NVM) is a problem with a lot of different approaches. Writing to NVM is often more expensive in terms of time and energy compared to writing in regular (S)RAM. The most used solution is checkpointing the code, where at each checkpoint all volatile data is written to NVM. Checkpoints can be placed recurring code like loops and at specific points in threads. However it is not always needed to write to NVM, when the energy storage is still full enough to reach not only the current checkpoint, but also the next. In case of such an excess of power, the device can decide to abstain from writing a complete checkpoint to NVM.

2.1.6 Sensors and Peripherals

Sensors connected to a batteryless system cannot use significantly more energy compared to the microcontroller. Doing so will disable the ability for the system to correctly respond to energy availability. Also sensors which take a long time to gather information are not always suitable for intermittent operation. One example of such a sensor is a gas sensor which heats up a part to determine the gas concentration in the air. Although efforts are made into powering accelerators as well [29].

2.2 Software

Even though there has not been created a single platform, effort is made into creating different kinds of software. Different aspects important for battery free operation are described below.

2.2.1 Memory Persistency

For battery-free devices is it important for the memory to be up to date. If memory is stored for longer periods of time, information can become stale.

For example, if a sensor measurement indicates a flooding and right after the measurement, the system is shut down for multiple weeks, unable to send out the information in the meantime. When the system boots up again, it will try to send the information about the flooding to other systems. At the time of sending this information is already out of date.

2.2.2 System and Peripheral State

At the moment power is disconnected from the system and its connected peripherals, also the state and configuration of these systems is lost. Most peripherals need some form of configuration in order to operate in the desired behavior, even for simple sensors. These configurations will often be lost after a power cycle and need to be restored efficiently for the system to operate correctly. In case the system takes too long to reinstate the correct configuration of a system, valuable energy and time can be lost.

2.2.3 Debugging

Creating bug-free code for sensor centered devices is already a difficult problem to solve. To help with this task are loads of different testing and debugging tools available. Spotting and recreating bugs for intermittently powered systems is even more difficult, due to the frequency with which the system is turning on and off. Regular debugging tools either require a device to be turned on during use or actively inject power to a system. Since the fluctuations in power can change the behavior of a system significantly. Therefore new debugging systems need to be designed in order to test devices without interfering with their power states or inputs. Different approaches addressed this issue went in the direction of analysis tools [27]. Solutions with hardware centric approaches like Shepherd [11] or Ekho [18], where energy harvesting logs can be replayed on a device in order to recreate buggy behavior.

2.2.4 Energy Aware Scheduling

Systems which are highly dependent on the amount of energy present at a given time, need to spend this energy wisely. The amount of energy consumed is not only determined by the peripherals attached to the device, also dependent on the energy spend on computing. In order to make sure the system does not have to recompute the same task multiple times, which wastes energy. The task can be split up into multiple smaller sections. The main approach in doing so is by utilizing checkpoints. These checkpoints either store the entire, or at the least the most important parts of the current execution. This information often consists of the program counter, the stack pointer, (parts of) the stack and peripheral states. Making a checkpoint costs time and energy, things that are precious for described devices. Determining when and what to checkpoint is an integral part of energy aware systems. One of the most studied approaches is to utilize an operating system to create checkpoints based on the scheduler, tasks and instructions. Example kernels are tinyOS [24], Chain [3], Mayfly [20] and Ink [40]

Another approach is done with checkpointless systems, like Alpaca [25] and Chinchilla [26]. These are compilers which checkpoints C code with minimal interference of a programmer. This is done by changing the intermediate code and inserting checkpoints on logical positions of the code, e.g. after a loop iteration and at the end of a loop. The advantage of this approach is minimal to no interference of the programmer on the code, even libraries which are not specifically written with checkpointing in mind can be compiled to do so. However, when operations are dependent on external hardware, the registers of this Another solution, which reduces the compute time of a task, is to change the accuracy of a result based on the amount of energy available at the current moment. Changing the compute time this way, could prevent the need for Non Volatile Memory. The main drawback for such an approach is the need for an algorithm which can actually take the energy level as a parameter when computing the result. As well as a tuning for each specific system, based on the total and current energy levels of the device. An example of such and approach is shown in [1].

2.2.5 Communication

Communication between intermittently operating devices is one of the most difficult aspects. The first biggest hurdle is to make sure both devices are active at the same time. Due to the lack of consistent energy supply or a guaranteed large energy storage, are devices turned off most of the time. During these moments a device can neither send, nor receive any data. Therefore, two devices which want to communicate with each other, need to find a moment at which they are both powered on. This bootstrapping problem has been looked at for entirely intermittently operating networks in [12].

After this first contact another hurdle appears, how make these devices sure they are active at the same time for the next communication time slot. As mentioned before, due to the lack of large energy storage and consistent energy delivery is there no guarantee both devices are turned on at the next communication interval. Also is the keeping a synchronized clock between the two devices difficult, when the clock is turned off at the same time as the device itself. Therefore, need both systems to have a robust a timing system, which we already discussed in section 2.1.4. One approach of staying in connection is done by continuously tracking and adapting parameters of an offline created model, like Bonito [13]. This implementation enables the ability to communicate between battery free devices. However, this custom implementation has no fail recovery, after a single missed interval the connection is considered lost. A completely new connection needs to be established after such a fail. Also is, due to the clock drift of the crystal, the maximum interval five seconds long. Such a short interval is too short for devices of which the power generator is based on the sun.

The problem with aforementioned solutions is the inability to operate with existing network technologies. For real-world solutions, interoperability is a must, even if devices are not made by the same manufacturer. Previous research made efforts to operate as a full Bluetooth peripheral device [7] or as a Bluetooth Low Energy mote [10]. The solutions are one-to-one or on-to-many communication, relying on a single host to operate the network. In building automation this would mean, placing lots of host devices in between the sensor devices. Resulting in extra overhead and still the need to design such a device.

2.3 Adjacent technologies

Transient computing is different from Intermittent computing due to the lack of an energy buffer. With intermittent computing the harvester stores energy in a buffer, often a battery or a capacitor, and the buffer is used when an interrupt if given. With transient computing the energy is not stored at all, but rather directly consumed by the hardware, like RESTOP [33]. The direct consumption adds extra complexity to the system (eg. decreased predictability, power generation equivalent to or larger than consumption of the system, more difficult time keeping). This type of computing is strongly related to the RFID roots. Where the responses are generated based on the incoming data of the device which also powers the RFID based device.

Energy- or Power-Neutral systems, are systems which do store energy gathered over time. In contrary to other discussed systems, their harvested power is equal to the power consumed over a specific amount of time. For example with photo voltaic based systems this time is often 24 hours. These type of devices have three different variables which can be configured, their harvested power, by changing the amount or efficiency of the harvester. Secondly, the total storage capacity of the device can be in- or decreased. Lastly, the power consumption of a device can be changed. Especially the moment of consuming the power is important, energy aware scheduling can greatly increase the productivity of a device. When the energy storage is full and new power is still generated, this access energy can be used to compute without having any negative effect on future operations. Coming back to energy aware scheduling, as discussed in 2.2.4

The different aspects of battery-free devices is endless. We are focused on the communication part of this problem. As discussed in 2.2.5 are solutions already present, however none of them is usable for building operation. Either the implementation is completely custom, making operating with other devices a lot more complex, or the communication protocol is based on a one-to-one principle. This makes operating in a larger network a lot more complex. Our solution is based on the Thread protocol, which is build from the ground up for building operation.

Chapter 3

Thread

Thread is lossy networking protocol intended to be used in the building automation industry. Due to the use of IPv6 and UDP as the base for its implementation, the network can interoperate excellent with existing Ethernet based networks. The combination of these technologies result in a low-latency low power mesh network. The OSI model of Thread is shown in 3.1. The lower two, Physical and Data Link, layers are based on the IEEE 802.15.4 Standard [21]. On top is the Network layer, based on 6LowPAN [2], which provides the addressing and routing. With the previous IP layer, can the User Datagram Protocol [31] be stacked. UDP is used within the Thread network to update information between nodes in the network. For ease of implementing application layer programs, Transmission Control Protocol [8] is supported as well. For indicating successful connections between devices the Internet Control Message Protocol [32] is used as well. By default, the Mbed TLS [39] library is provided as well, for encryption. Mbed TLS can also be utilized by the programmer in the application layer.

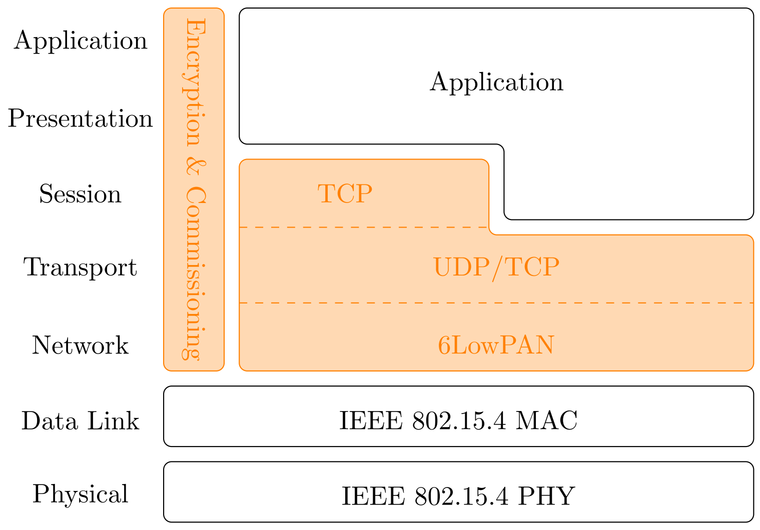

Figure 3.1: **Network stack overview of Thread.**

3.1 Roles

Depending on the capabilities of a device, said device can have one or multiple roles within the network. In this section we will go over different roles and expected capabilities for these roles.

3.1.1 Full and Minimal Thread Device

The first major division between different Thread devices, is the distinction of Full and Minimal Thread Devices. A Full Thread Device (FTD) has all the capabilities of a network baked into the firmware. FTDs are therefore capable of routing packets through the Thread network. With these extra capabilities, an FTD can operate as an end device as well as a router. When the FTD is operating as an end device it is referred as a Router Eligible End Device (REED). At the instance where an extra router is needed, a REED can be upgraded to a Router. More information about routers will be given in Section 3.1.2.
The second device type is a Minimal Thread Device (MTD). These devices do, as the name implies, only have the bare minimum firmware to operate on the Thread network. This reduces their hardware requirements of volatile and non-volatile memory as well as a reduction in power consumption, since their radio does not need to be turned on at all times. However, operating in such a way mandates an MTD to operate only on the edges of the network as an End Device. In Section 3.1.3 more details about these devices will be given.

3.1.2 Router

Routers form the main backbone of the Thread network, together with other Routers they create a mesh network to relay packets around to their correct destination. To make the routing possible, router devices are expected to have their radio turned on all the time. This will make it possible for End Devices to send their packets without checking if another device is able to receive its data. Also does the router node need to keep a list of all its connected End Devices, such that it can store and relay packets destined for the End Device. The Router node will send the stored packet at the moment the End Device is active and able to receive its incoming packets. Statistics about the connected End Devices need to be stored by the Router node as well. The statistics will determine when an End Device is released from the network, how often it is expected to reconnect to the network. Thread will try to keep the amount of routers inside a single network between 16 and 23. The maximum number of routers is 32.

Border Router

Special version of the Router node is a Border Router. The Border Router has next to a Thread interface also a connection to another network, for example a Wi-Fi connection. The dual interface gives the device the opportunity to act like a firewall between the two. Packets from the one network can be forwarded to the other network and vice versa. Which packets are let through this firewall depends on the rules set on the Border Router. Inside a Thread network can be multiple Border Routers. The choice for accepting more than one Border Router is to prevent a single point of failure inside the network.

3.1.3 End Device

End Devices are the leaf nodes of a Thread network. In contrary to Router nodes do End Devices not need to keep their radio powered all the time (in this case they will be a Sleepy End Device). Therefore End Devices are more geared towards battery(less) devices. End Devices can only be connected to Router node. Communication between a Router and End Device is initiated by the End Device. Since Routers are always listening for incoming traffic, packets send by the End Device are more likely to reach its destination.

A Full Thread Device can also be a Router-Eligable End Device (REED). A REED can change its role from an End Device to a Router node in case the network requires more Routers. This need is based on the amount of Routers present in the network (24 is the optimal number), as well as their connectivity. If the REED is the only device which can connect to a new End Device on the network. It will upgrade itself to a Router. Th reverse is also true, if End Device are lost from the network or the number of routers is exceeding 24, a Router can be downgraded to REED.

Sleepy End Device

Sleepy End Devices (SED) are a special version of the End Device. These SEDs are End Devices which do most of the time sleep. Starting from Thread 1.2 [38], the specification allows for Synchronised Sleepy End Devices (SSED). Contrary to regular SEDs, these devices return from sleep at a fixed time interval. This interval is known by the accompanied Router node, which will send buffered packets at these intervals. By sending at this interval, the End Device does not need to send a request for the buffered packets anymore, reducing its on-time even further.

3.1.4 Additional Functions

Next to the earlier mentioned Border Router, the Thread network has more additional functions which need to be fulfilled for a correctly functioning network. These functions are explained below.

Leader

The Leader function inside the network is given to at least one and at maximum one of the Routers. A Leader is a self elected function, the router creating a new network is automatically also the Leader, every subsequent Router stays a regular Router node. In the case where the Leader is removed from the network, one of the other Routers will take up the role as Leader. The function of a Leader node is to distribute network configuration information to all the connected Routers. The Leader determines whether another REED needs to be turned into a Router or vice versa.

Commissioner

In order for more Thread devices to enter the network a Commissioner is need. This role can be fulfilled both on and off the Thread network. An on-mesh Commissioner is one of the Routers which distributes the network key to Joiner devices. An external Commissioner is also possible, for this a Border Router needs to be present which connects the external Commissioner to the network. An example of an external Commissioner is a smartphone with a special commissioning application installed on it.

The commissioning process works as follows. There are two devices, the Commissioner and the Child which wants to join the network. The Commissioner retrieves the passkey of the Child via a different medium (often a QR code is used). With the information from the QR code, the Commissioner instructs the Child to configure the radio the same as the rest of the devices inside the Thread network. Also is the network key distributed to the Child, with which it can successfully connect to the network. From the Router the Child is connected to will the Child receive the IP address(es).

3.2 Topology

A Thread network takes the form of a mesh network, where routers can have multiple connections between each other and end devices are connected to one router at a time. An example of the topology of a Thread network can be seen in Figure 3.2. In this example the gray cloud represents a network different from the Thread network, lets call this the internet. The bridge between the Thread network and the internet is the white square with blue border, this is the border router. The hexagons represent other routers within the network. The black hexagon fulfills also the leader role. As can be seen the router notes have multiple connections between each other. The circles represent End Devices, with the color indicating the type of End Device. Circles filled with a darker blue color represent Full Thread devices which are operating as an End Device. These nodes are capable of becoming a Router and are therefore called Router Eligible End Devices. The circles with a lighter blue color are Minimal Thread Devices which are operating as regular End Devices. The white circles connected to a router with a solid line represent Sleepy End Devices. And lastly, white circles connected via dotted lines to a router represent the Synchronized Sleepy End Device.

Due to the combination of creating a mesh network, the self elected leaders and the ability for nodes to scale up and down their function, is the network resilient and self forming. In the case where the Leader loses connection with the rest of the Routers, one of the other Routers in the network will take up the role as a Leader. The disconnected Leader thereby creates a new network, separate from the old ones. If no Commissioner is present, the Leader will also take up the role as the Commissioner, thereby completing all the necessary roles for creating a Thread network.

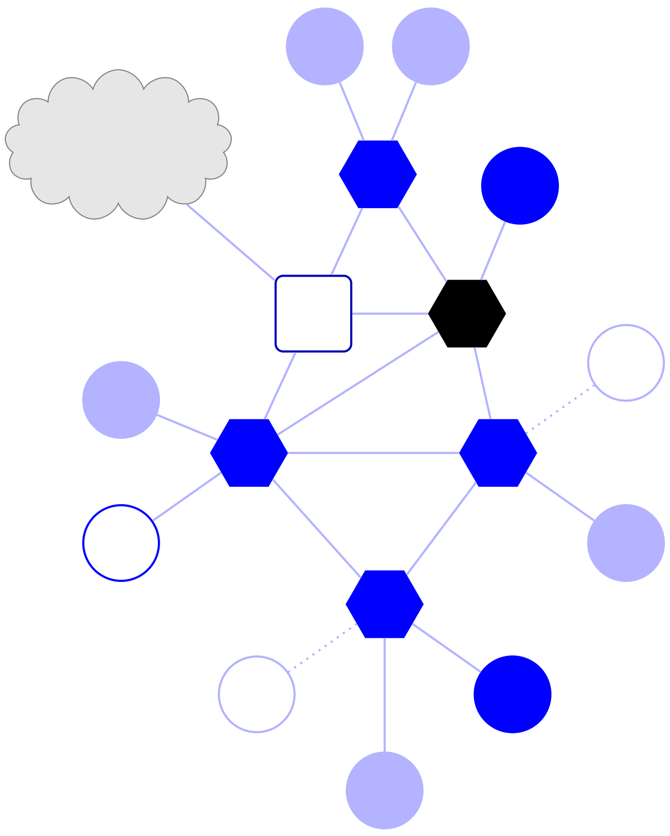

Figure 3.2: **Network topology example, square and hexagons are Routers, cloud represents a different network, circles are End Devices.**

3.3 Addressing

Thread utilizes IPv6 to address nodes inside the network, based on the 6LowPAN. Addresses are distributed to nodes both based on their location within the network topology as well as for their function.

The location based addressing can be divided into three scales, Link-Local, Mesh-Local and Global. In Link-Local addressing, addresses are distributed to a Router and all the devices it can communicate with a single hop. This includes the leaf nodes connected to the router, but also the other routers which are directly connected to the Router in question. Addresses are given based on the extended MAC address of the device. This type addressing is used for setting up connections between devices and configuring these connections. With Mesh-Local addressing, all devices within the same Thread network can be connected. This is the main way of communicating between devices within the mesh network. The address of a device is determined on their location within the network. Every device gets an identifier called the Routing Locator (RLOC), this number consists of both the location number of the Router as well as the number of the End Device. An example for Router number 1 and End Device number 1 can be found in Figure 3.3, this results in a value corresponding

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
Router ID						**R**	**Child ID**								
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1

Figure 3.3: **Construction diagram of the RLOC16 numbering, 0x401 in this example.**

to 0b0000010000000001, which is equal to 0x401. In the case where an End Device connects via different Router instead, the RLOC16 of this device will therefore change as well. The last scope is the Global addressing, all devices with an interface to a network outside of the current Thread mesh, get such an address. Devices do not need to be directly connected to an outside network to have an interface which connects to other networks. Addresses can be assigned by a DHCP server, via Stateless Address Autoconfiguration [30], or set in the application layer. These interfaces are used to communicate with devices outside of the Thread mesh network, even directly with devices on the internet.

Next to the location addressing, do devices also have multicast addresses. These addresses are used to send network change information around the network. These addresses are divided into both the scopes and groups, the two groups are only FTD and FTD with MTD. The scopes are either Link-Local or Mesh-Local. There are not addresses specific for (S)SED, neither are they included in any of the groups.

The ability to directly connect with services outside of the Thread network, is one of the main selling points of this network type. This enables manufacturers to not only let devices operate in their local network, however also connect them directly to servers of the manufacturer. Enabling the producer to give extra features to their devices based on the connection to their own servers. While still being able to operate within a smaller network, for example the local network of the building these devices are located in.

Chapter 4

Implementation

In this Chapter we will go over the implementation details of the Thread protocol used in this project. The implementation for Thread is not made entirely from scratch, the protocol is too complicated to do so. We used the OpenThread [14] library an open source implementation of the Thread protocol made by Google. Most hardware vendors,who make compatible radios, have taken this open implementation and ported the software to work for their own devices. Nordic Semiconductors has done it for the nRF52840 as well [15], the hardware platform of our choice. From this base implementation we changed none of the internal programming to build the intermittent version. The OpenThread implementation of the Thread network can be described as seen in Figure 4.1. The OpenThread library depends on the Platform Abstraction Layer (PAL). In this PAL interactions with the specific hardware are done, these contain a minimum set of functions which are needed for the device to operate as an MTD. More functions can be implemented for support to function as an FTD. Some Advanced Features on top of the FTD can be implemented as well, these include, but are not limited to Auto Frame pending, Energy scan with the radio and hardware acceleration for encryption. From these Advanced Features, Auto Frame Pending is needed for devices to behave as an SED or SSED, this advanced feature is implemented for the nRF52840. The functions needed for HAL are split between these pillars; Alarm, Serial (UART, USB, SPI), Radio, Entropy, Miscellaneous, Storage, Logging and optionally some platform specific functions. Below the PAL is the Hardware abstraction layer, which is implemented by the manufacturer. On the other side of the OpenThread Core Stack is the Application Layer, which operates based on the functions inside the OpenThread stack.

4.1 Platform Abstraction Layer

From the previously mentioned pillars, the Alarm, Serial, Radio and Entropy are important to look into. This subset is important, because they either use hardware registers which need to be set correctly upon restart or provide a basis for the encryption library. In encryption iterators are used to create the correct encryption chain (Entropy), these need to be restored correctly after a restart. UART and USB are not used in our experiments for the End Device. The SPI

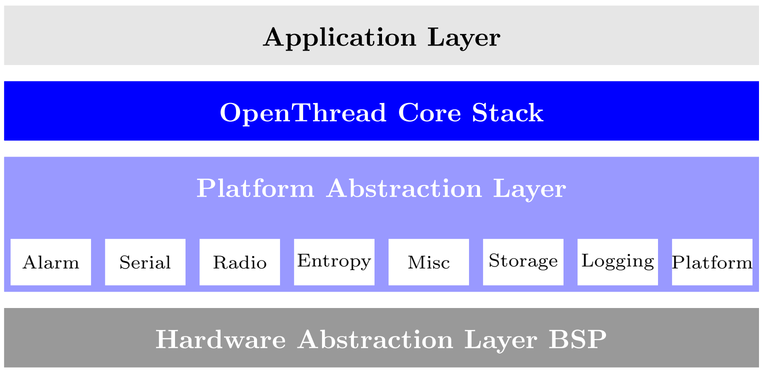

Figure 4.1: **Software stack of the OpenThread library, with the Application Layer on top and Platform Abstraction Layer below it. And the Hardware Absraction Layer on the bottom.**

is used and was looked into, the configuration of the pins are set at every boot. The reason for resetting the pins every boot, is to use the external RTC, which is needed to align the internal RTC.

For the Mbed TLS library is the Entropy part of the microcontroller to set an initial seed value of the software random number generator. After setting this initial value the hardware Entropy functions are not used anymore. By setting the Mbed TLS library in full software mode, without any hardware optimizations, will it store all values in memory and important information is thereby stored in FRAM before powering down.

On a single device, it is possible to use multiple Thread capable radios. Storing the information about the radio and configuration is therefore important when building a packet. The whole OpenThread library is made with this capability in mind. Some information about the hardware layer is therefore also stored in the configuration of the class which manages the radio. When sending out a packet, the configuration of the radio is checked before sending. If the configuration is incorrect, OpenThread will instruct the hardware to take a correct configuration. Even when the radio is in the incorrect configuration, because of for example a restart, it is still configured correctly before sending. And, since the information about the correct configuration is stored in memory, this information is also stored in the FRAM before powering down. Thereby ensuring the consistency of the radio, even after a power cycle of the system or the radio.

In the Alarm pillar of the HAL, are all the function calls to the RTC located. For the device to operate properly it needs correct timekeeping, even between reboots. This timekeeping is based on the implementation by Winkel, et. al. [7]. Where strong synchronization is needed between the external RTC and internal RTC. With this implementation sub millisecond synchronization is possible between the two real time clocks. The synchronisation interval is calculated by finding the lowest common denominator of both the clocks frequencies. During these intervals, the amount of ticks per clock is always the same (although different for each clock). Since the exact interval value is known, multiples of this interval can also be calculated and stored for the next power up. This sub millisecond synchronisation is especially important for devices which do also synchronise with their Parent node, such as SSEDs. During none of the

SSED tests was the connection lost or a request needed to be resend. Thereby confirming the usability of this approach.

4.2 Application Layer

On top of the OpenThread Core Stack is the Application Layer located. This layer gives the developer the ability to interact with the network stack directly. Here, the configuration variables can be set, callback functions set and functions called to send out packets.

Some of the configuration settings used are for setting an MTD as an (S)SED. For example the poll period can be set, this is the time the Router waits between two requests for data from the End Device. A timeout value can be set as well, with a minimal value of 62ms and a maximum value of 4.25 years, according to the specification [17]. These configurations are set upon the first boot and stored in memory for every intermittent sequential boot. The OpenThread core requests the correct configuration from the Router node when it becomes available. In the Application Layer is also tested whether the device is correctly connected to the Thread network. Only when the device is correctly connected, will it be allowed to entirely shut down. The library is set up with its own instruction stack implementation. On this stack will it create and store instruction, waiting for execution until a specific function in the main loop is called. This instruction will go through the entire stack of instructions and execute them all. To prevent instruction from being stacked for too long, the stack is checked to be empty before shutting down as well.

4.3 Transferability

The transferability of this project can be viewed from multiple different aspects. The aspects discussed here are, swapping out the OpenThread to new version of the library, swapping out the hardware to a completely new platform, or changing out the protocol.

At the moment when a new version of the OpenThread library is published, the amount of effort needed to put into transferring the current code base to the new version completely depends on the OpenThread library itself. In the case where the new version is fully backwards compatible with older versions, no changes need to be made. This makes it a lot easier to upgrade to a new version. The reason for this ability is the fact that all changes made to make it run intermittently are only in the HAL and Application Layer of the project.

Porting the code base to a completely new platform, depends on whether the implementation for regular operation is already present. In the case where the manufacturer of the new platform already implemented the HAL with the OpenThread network, only the changes for operating intermittently need to be made. The changes for intermittent operation in the Application layer do not need to be adapted, however changes in the HAL and for the external RTC might need some changes. The main hurdle is the synchronisation of the external RTC with the RTC internally in the microcontroller. Also might it be needed to change the algorithm which determines the next wake-up time. The algorithm is closely related to the hardware counters in the microcontroller.

Changing out the protocol for another is not possible without throwing everything out and starting over. The changes made are specific to the function calls of the OpenThread framework. Even changing the library out to another, non OpenThread, Thread library, has little chance to be compatible. The function calls need to overlap significantly in order to match. The combination of specific properties makes Thread specifically well equipped to let device operate intermittently. The ability to make devices operate only as leaf nodes in a network (End Device) and thereby offload most of the routing work to other devices. The ability to stay disconnected from the network for a long period of time, without the need to completely reconnect every time. The ability to connect at a fixed interval

4.4 Connecting to the Network

In a regular Thread network the joining device requests access to the network via the Commissioner, which provides the network wide key. In order to ease the network creation process, the key is stored in the configuration when programming the devices. This is the case for both the Router (FTD) and the End Device (MTD). Although the credentials are stored, the connection between the FTD and MTD still need to take place. If this is the first time the MTD connects to a Router in this network, the information about the MTD needs to be stored by the Router. Since the Router determines when an End Device is kicked off the network, it needs to store the last time it received a message from the End Device. Also are the age, RLOC16 (for routing purposes), extended MAC and information about the type of device stored. The second connecting type is when the device is reconnecting to the network it already accessed before, this might happen when the End Device runs out of energy and does a cold start. If the device is not kicked off the Thread network by the Router yet, it will make a quick reconnect. This form of (re)connecting is a lot quicker, since the MTD does not need to send its preferred settings to the FTD anymore. Therefore, reconnecting before the connection timeout happens, is more efficient compared to making a cold connection.

4.5 Leveraging Thread for Intermittency

One of the major advantages of using Thread is the ability to be sparse with the amount of packets sent in the network, without the device sending out the packets being removed from the network. An End Device can be operating in three different categories, each with their own way of communicating.

- Default configuration for an End Device, Naive (A)
- The default configuration with basic improvements, Semi Naive (B)
- Sleepy End Device (C)
- Synchronised Sleepy End Device (D)

The first is the default configuration for the End Device, **Naive** (A). The device has its radio powered on at all times, like the Router. Therefore it does

Property	**Naive (A)**	**Semi Naive(B)**	**SED (C)**	**SSED (D)**
Radio Always On	Y	N	N	N
Goes to Sleep	N	Y	N	N
Turns Off	N	N	Y	Y
Polling	N	N	Y	Y
Listening Only	N	N	N	Y
FRAM	N	N	Y	Y
External RTC	N	N	Y	Y
Δt Messages	4.25y	4.25y	18.6h	10.5s

Table 4.1: **Device type configuration overview.**

not need to set a poll time, since the device is expected to send and receive packets at any time. This type of device is intended to be used with an "unlimited" energy source, connected to the power grid. In this setup the Router does not buffer any messages for the End Device. Although it is possible for this device to request a poll time from the Router, this function does not have any beneficial implications. The poll will only add extra unnecessary communication, the device can send and receive at will to and from the Router. Also does the base implementation have none of the basic energy saving configurations implemented. For these energy saving we used the following instructions __WFE, __WFI and __SEV. The __WFE instruction sets the hardware in a state where it will wait for the next event. During this wait period most other systems are not operating, thereby reducing energy consumption. The instruction is often used in combination with the __SEV, which sends out an event. This instruction is used where the hardware does not send out an event by default. The __WFI instruction is the most effective way to reduce power consumption. The system will power down as much as possible, until it receives an interrupt, which can only be given from a small number of peripherals (including external pins and the internal clock). In contrary to the Naive (A) implementation, are these basic energy saving configurations enabled on the **Semi Naive** (B) implementation. In Table 4.1 are properties, which differ between implementations, displayed.

The second option is a **Sleepy End Device** (SED, C), this device has, in contrary to the first option, most of the time their radio turned off. Only when expecting a potential message, or sending one out, is the radio active. This device sets a poll period, which indicates the time between polling the Router node, to which the device is connected, for new packets. The End Device dictates this period. If the Router has any messages buffered for the End Device, it will send the buffered messages right after the poll. The maximum value of the period is constrained to approximately 18.6 hours [16].

The last option discussed, is the **Synchronised Sleepy End Device** (SSED, D). This device synchronises the time on which it will turn on its radio with the Router node. When operating in this mode, the SSED does not need to poll the Router before receiving information. This type of implementation is best used for End Devices which send less often compared to receiving information. Especially is the response time is needed to be small. The longest time for this feature, called Coordinated Sampled Listening (CSL), can be configured to be approximately 10.5 seconds [16].

All of the mentioned intervals are used to create time slots for devices to

Use case	Interval	No connections per day
1. Thermostat	10 minutes	144
2. Valve Controller	10 seconds	8640
3. Flood Sensor	30 seconds	2880
4. Window Sensor	1 day	1

Table 4.2: **Number of times the use cases connect to the network.**

operate. It is always possible for messages to get lost. However the difference between getting completely out of the scope of the Thread network and losing a messages get faded. To prevent devices from reserving resources for too long, is it possible to also set a timeout value. If no connection is made before the timeout timer reaches its end, the device is removed from the Thread network. The value for this maximum timeout value can be requested by the End Device. The maximum value of this timer is approximately 4.25 years [16]. The default implementation needs to minimally send one message to the router within this period.

4.6 Use Cases

Building a working application on top of the Thread stack falls outside the scope of this project. Nevertheless, with the gathered power consumption data, power estimations of a few example use cases can be made. In order to demonstrate multiple behaviorally different implementations, the following use cases are made. For all use cases are sensors or actuators not counted towards the energy consumption of the devices, due to the large difference in power consumption between different sensors and actuators. Each of the discussed use cases have different connection properties, which will be explained per use case. An overview of the different properties is given in Table 4.2.

The first use case is a device which does only need to gather some date from its surroundings, a thermostat. In a smart home, controlling the climate control on a room by room basis decreases the power consumption of a home, compared to centrally operated home. In order to control a room a feedback loop with temperature and possibly humidity control is needed for the operation. The thermostat in this scenario reads the temperature and humidity data four times per hour, since climate control does not need a high interval reading, the measuring interval can be relatively long. After a reading the device will send the retrieved data out and do nothing until it has to sense for the next moment. For this use case we will assume an inter-measurement time of 10 minutes. For an entire day this amount to 144 times of booting up and sending out the measured data to the Router.

The second implementation is a device with a small actuator, operating with only a few times a day combined with a sensor. For this use case we take a valve controller for a heater. The central heating system of the building pumps warm water throughout the building and per room there are one or more heaters installed. These heaters are operated by a valve, which is controlled with our controller. This implies the heater does only respond to a temperature measured at another place in the room. So the controller does not send information out for itself, it only responds to messages received by the measuring device. Since the heating needs to stop as quickly as possible after a set temperature is reached, the valve controller still needs to operate within a small time window after the message about the new temperature is send out. This same valve control can be applied to the main water valve of a building, to prevent flooding, or for plant irrigation. The device will check every 10 seconds for a new message, 6 times per minute, 360 times an hour or 8640 per day, ensuring quick responses even when a flooding is detected.

The third example use case is a water flood sensor. This type of sensor is often placed near a boiler, washing machine or dishwasher to detect a leakage of these devices. This information needs checked often, the leakage need to be detected before the water can do any damage. The device also needs to send the information over quickly, for the owner or other systems to respond accordingly. If any of the forementioned machines is operating, the sensor needs to keep running and communicating with other systems to exclude water damage. To make sure the device is always operational and sending out its last information the device will report its state every 30 seconds, a full leak needs to be detected within a minute to prevent major leaking issues. Sending out this often will result in 2880 messages per day.

Another use case is a window scatter sensor. This sensor will send out information when the glass it is connected to gets shattered due to for example a break in. This sensor does not need t update its state so often. The bare minimum is communicating to prevent it from getting kicked off the network. Since this is less than one message per day, the amount off messages it needs to send out maximum is when the device either detects the scattering of glass or when the device needs to make sure it stays connected. The maximum number of messages per day is in both cases one per day.

Chapter 5

Implementation Analysis

In this chapter the testing of the developed system is described. First is the test setup described, on both the hardware and the software side. Next do we take a look at the produced results, followed by the impact of these results on our use cases.

5.1 Test Setup

The intended goal is to gather information about the power consumption of the device in different operating conditions. with a focus on the difference between a regular implementation and an intermittent operating device. In Section 3.1, we discussed different conditions in which a Thread device can operate. From these different roles, we created the following four implementations to test. Implementation A: the basic Naive implementation. Implementation B: an improvement on the previous implementation with minor energy consumption improvements, called Semi Naive. Implementation C: fulfills the role of Sleepy End Device (SED), operating in an intermittent way. Implementation D: a slightly different role, a Synchronised Sleepy End Device (SSED), also operating intermittently.

In Section 4.6 a few use cases are discussed. Since implementing these use cases is outside the scope of this paper, the power consumption of these implementations is approximated, using the gathered data in Section 5.2.

5.1.1 Hardware

At the heart of the hardware setup is the mote created for the research of FreeBie [7]. To program the mote the Segger J-Link [35] is used. Power for the device is provided by the Power Profiler Kit II by Nordic Semiconductor [37]. This Power Profiler provides 2.7 V to the mote, making sure the device is always provided with enough power to start operation. The Power Profiler Kit II can detect a range from 200 nA to 1 A at 5 V and an accuracy of ±10%. The current consumption measurement is low enough to detect the feasability of the implementation. The power is injected on the test points VBAT and GND using probes. The mote will be programmed as an End Device, it needs a Router (and Leader) node to join the network. This role is fulfilled by the

Nordic Semiconductors nRF52840 DK [36], powered directly over USB. The distance between the devices is set to 20 centimeters.

5.1.2 Software

For the software we started out with the default implementation given by Google in association with Nordic Semiconductors [15]. This implementation gives the option to build the system for a Full Thread Device (FTD) or a Minimal Thread Device (MTD). The Router and leader is build using this default implementation, without any of the intermittent changes, as an FTD. The mote is programmed as an MTD with modifications for intermittent operation. Important to note, all modifications are made in either Hardware Abstraction Layer of OpenThread or the Application layer on top of the OpenThread implementation, as described in Section 4. The poll period is set on a 5 second interval, also is the radio configured to turn off when OpenThread is idling. A timeout, the time before getting removed from the network, is set to 11 times the poll period, 55 seconds. The version without intermittent implementation of the MTD is used as the Naive implementation (A).

To configure the Semi Naive implementation only the functions __WFE and __WFI (described in Section 4.5) are added at a point at which the device can afford to wait for the next interrupt. This point is the same as described for the intermittent implementation. With these functions the nRF52840 turns off most parts of the device down, based on which parts are not need to wait for the next interrupt. These optimisations are implemented by the manufacturer and build into the hardware of the microcontroller.

Intermittent Implementations

The Sleepy End Device (SED), implementation C, and Synchronized Sleepy End Device (SSED), implementation D, are both full intermittently operating devices. At the point where the Semi Naive (B) implementation goes to sleep, the SED (C) and SSED (D) instead test whether it is possible to turn off. To prevent the system from turning off when it still expects to perform work or to receive a packet from the Router, some checks set into place. The most important check is whether the device is already connected to the Thread network. During the connection making process, the device needs to respond quickly and receive multiple packets with information about the network configuration. Therefore it is unwise to turn the system off before the connection is established. The second check is by utilizing the OpenThread task stack to check if there are any tasks left. When there are still tasks left in the stack, the system will not turn off. This ensures a quick handling of all outstanding tasks. Also is checked whether the radio is in sleep mode. After use the OpenThread library is configured to let the radio sleep as soon it is not used anymore. This makes sure the system is neither sending out any packets, nor expecting to receive any packets. The last check is the alarm, the alarm can send out an interrupt without adding a new task to the stack, while setting another flag. At this moment the system is expected to handle the set flag within reasonable time. The system could turn off for multiple seconds, longer than what is seen as a reasonable time frame. Only if all these checks are passed is it possible to schedule the turnoff time.

Next the system will prepare for shutting down. Before being able to shut down, another check is needed. Whether there is still enough time left to turn off, is calculated based on the hardware timers inside the microcontroller. These RTCs store timers for different purposes, the timer with the shortest time left is found. If the shortest timer will fire before the system is able return from turning off, the device will not shut down. When there is enough time left, the system will turn off the radio, store the current memory in FRAM and an interrupt timer is set on the external RTC. At the moment of setting the timer on the external RTC, the power to the system will also be removed.

When turning on again, some of the boot up code is setup to at least be able to read out the external RTC and FRAM. The last stored breakpoint is loaded from the FRAM and a timer is set and another timer is set on the external timer, with the intention to start the internal timer of the microcontroller at the synchronisation point. This synchronisation is done, such that the timers, which were set before the shutdown, can be reduced with an exact amount of ticks. Also does this tie all the timers used in the OpenThread stack (called Alarms) to the external timer, resynchronising after each intermittent boot.

The full restoration of the memory, even up to the correct stack pointer, ensures all information about the state of the networks stack and previously stored information is recovered. With all the counters and keys are recovered, the encryption library (Mbed TLS [39]) is able to encrypt and decrypt the next packet as soon as the SRAM is restored.

Both the SED and SSED work with the configuration described above. However their operation is a bit different in the Thread network. The SED sends out a message to its connected Router at a fixed interval, polling every time for new packets. While the SSED does not send out any messages, instead just turns on the radio on at a specific time interval and listens for incoming packets. In this setup the SSED needs to confirm from time to time it is still alive.

Connecting to the Thread Network

For various different reasons a device can lose connection with a network, for Thread end devices this can happen in two different ways. These two different connection loses do also behave differently when reconnecting to the network. As described earlier, creating the connection with the network is not part of the intermittent operation. However we are still interested in the amount of energy it takes for a device to connect to the network. This information might be needed in order to determine the amount of energy storage needed when designing a new device. The two different ways of establishing a connection are described below.

At the moment of the very first boot, or when the end device is removed from the network. It needs to reestablish a connection and be verified inside the network. This process can take a long time, relative to sending and receiving single packets. The amount of data send back and forth is large and some devices even need to synchronise their clocks between each other (SSED (D)). The type of boot sequence is also referred to as *reset start*.

The second boot type is where the end device lost the configuration it previously had, while the network still has the data about this end device stored. In this case only a small amount of information need to be send from the Thread network to the end device to synchronise again. This moment can occur when an intermittently operating device ran out of all of its energy, where also the external RTC is not powered anymore, or when the checkpoint is invalidated. This type of boot sequence is also referred to as *soft reset* or *soft start*.

Optional Optimizations

Even though the distance between devices is set to only 20 centimeters, the radio on both devices is set to the default signal strength. The option for adaptive power is not enabled to prevent fluctuation in the data. In practice this configuration could be changed to reduce power consumption to a minimum.

Parsing the Results

After the power consumption data is retrieved, it needs to be parsed for determining the power consumption of the use cases later. The use cases are described in a form of the amount of times the device needs to connect to the Router node. The power data is therefore split into peak and low power consumption parts. With the peaks being the moment the device needs to communicate with the Router. The low power data describes all the time when there is no peak. In most of the implementations, this is when the device is sleeping or turned off (B & C & D). For the Naive implementation this difference is less clear, as we might see in the results. However the edge of high and low is set slightly above the average power consumption.

For the booting procedures do we define the duration of the boot sequence as the time it takes from the first power consumption, after reprogramming or restarting, until the first low period. In this period a first regular information packet is already included. This first regular sending of packet does not include the restart process of an intermittently operating device (it misses the reading from FRAM) and should therefore not be included as one of the regular packets.

5.2 Results

In this section the results of the experiments are shown.

5.2.1 Power Consumption During Regular Operation

The first result shown in Figure 5.1 shows the current consumption in mA, on the y-axis, for the four different implementations (Naive, Semi naive, Sleepy End Device (SED) and Synchronized Sleepy End Device (SSED)) over a time span of 120 seconds. The part where the device connects to the network is removed and will be looked at in Section 5.2.2. We choose to show a time span of 120 seconds since this is the timeout of the SCL setting of the SSED node. The graph shows a significant power consumption difference between the Naive implementation and the other three. Peaks are visible in the consumption, which correspond to the sending and receiving intervals of the nodes. The device needs power to receive and decode the radio waves, resulting in a higher current consumption. These peaks are occurring with an interval of approximately 5 seconds, matching with the poll period of the nodes.

In this graph, the difference in power consumption between the Naive (A) (top left graph) implementation and the other three is clearly visible (B & C

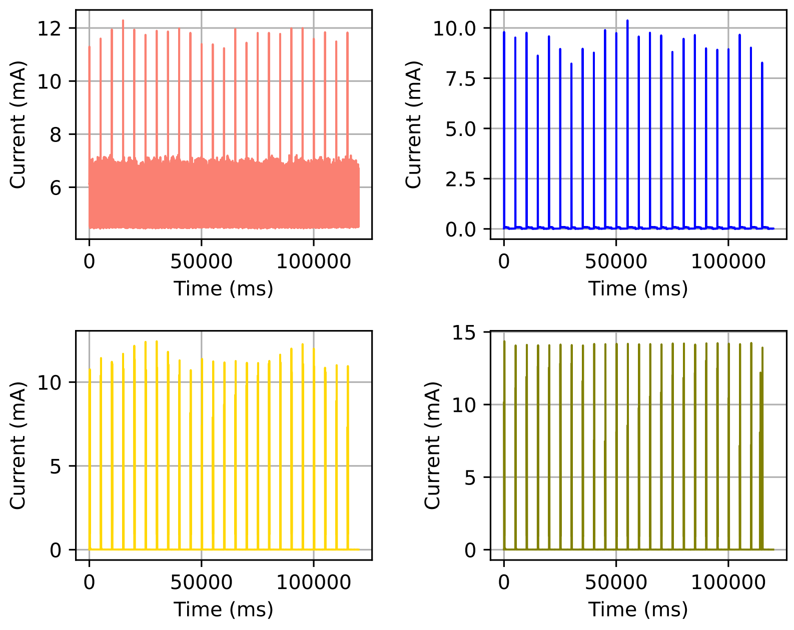

Figure 5.1: **Power consumption of the device during the first 120 seconds in different configurations. With the implementations, top left Naive (A), top right Semi Naive (B), bottom left SED (C) and bottom right SSED (D).**

& D). The graph of Naive never reaches a current of less than 4 mA. Later we will take a closer look at the average power consumption per state. The results clearly show Naive has the highest power consumption overall. Comparing the peaks with each other, seem the tip of the peaks to be highest with SSED (D), followed by Naive and SED (A & C) with comparable peak heights, the lowest peaks can be seen with Semi Naive (B). Take into account, this is only a subset of the entire data set, so the peaks can differ over time. Also does the width of the peak a significant impact of the total energy consumption (more about the width when discussing the second Figure 5.2). Due to the 5 second poll period or SCL interval, do all devices turn on with an interval of 5 seconds. This is also visible in the power consumption graph, the regular interval of the peaks are spaced 5 seconds apart. However, the SSED does have one extra peak at the end, right before the 5 second interval. This is the message sent from the SSED to the Router. The SSED does normally only listen for incoming message, without any polling message. The Router does not receive any messages if it has nothing to send to the SSED. Therefore does the SSED send a message to show it is still online.

The second graph, shown in Figure 5.2, zooms into one of the five seconds poll period for every node. Here, the peaks are more distinct compared to the previous graph. The graph shows a shorter power pulse for the Naive and Semi Naive (A & B) approach, while the SED and SSED (C & D) take a bit longer to finish. Later in this section we will take a closer look at the power consumption peaks in Figure 5.4. As mentioned before, the Naive implementation performs the worst when the device is waiting for the next action to perform. When waiting for the next operation the Semi Naive implementation seems to have a longer tail before reducing the power consumption. However, this is not visible in this graph.

Figure 5.3 shows a detailed view of one of the off-periods. In these figures the x-axis indicates the time, while the y-axis is different between sub-figures, showing the current consumption. In these graphs is the difference between the implementations clearly visible. The Naive (A) approach consumes large amount of power. While the Semi Naive (B) implementation a big improvements shows compared to the Naive implementation, is it less consistent and shows less performance compared to the real intermittently operating implementations (C & D).

Figure 5.4 presents the regular operation of the different implementations, taking a closer look to the individual peaks. The time interval is set to 250 ms on the horizontal axis, with the complete peak(s) shown. Important to note is, what first seem to like a single peak for SED and SSED (C & D), are actually two peaks close after each other. This is due to the reading and storing of information to and from the FRAM. This shows the overhead incurred from using the hardware needed for intermittent operation. The time from approximately 75 ms to 175 ms for SED (C) and from 75 ms to 200 ms for SSED (D) relatively, show a long delay from the moment booting seem to be finished, until the sending of a packet and storing the checkpoint.

5.2.2 Boot Power Consumption

As described in Section 5.1.2, there are two different boot sequences. Both sequences reboot the system because the checkpoint stored in FRAM is con-

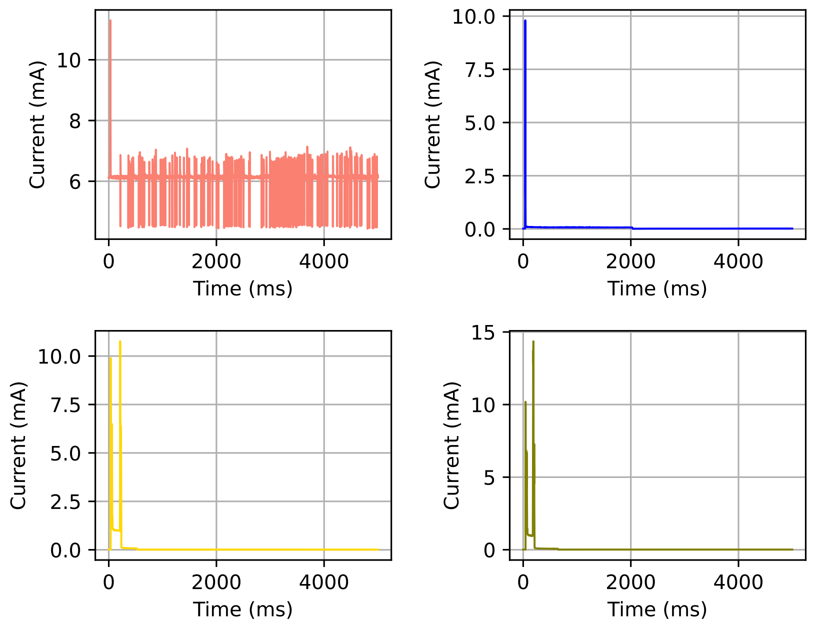

Figure 5.2: **Power consumption of the device for five seconds in different scenarios. With the implementations, top left Naive (A), top right Semi Naive (B), bottom left SED (C) and bottom right SSED (D).**

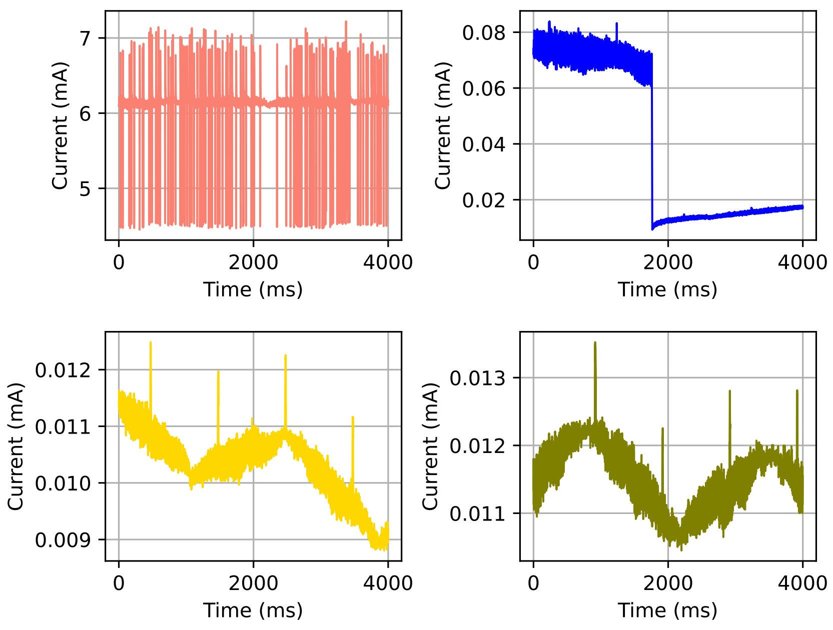

Figure 5.3: **Close-up of the low power consumption during off periods. With the implementations, top left Naive (A), top right Semi Naive (B), bottom left SED (C) and bottom right SSED (D).**

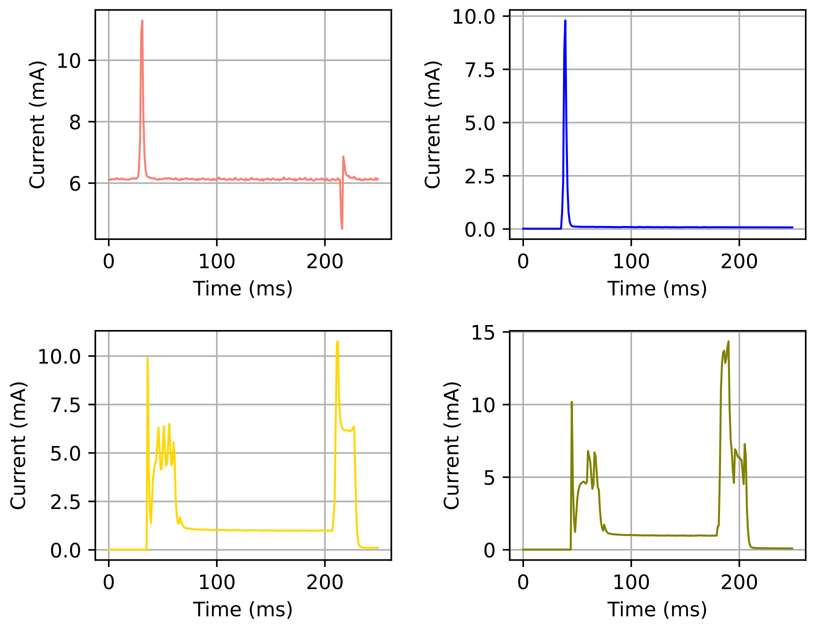

Figure 5.4: **Close-up of the peak power consumption during sending and receiving packets. With the implementations, top left Naive (A), top right Semi Naive (B), bottom left SED (C) and bottom right SSED (D).**

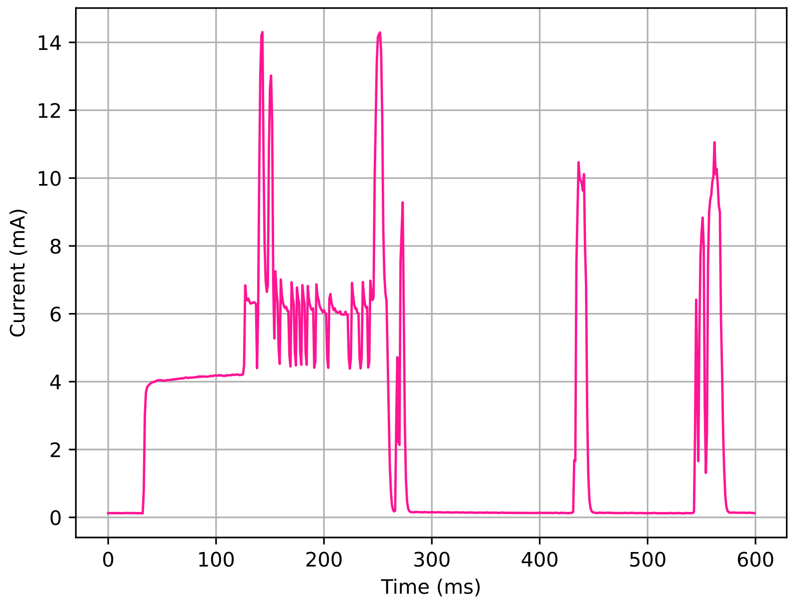

Figure 5.5: **Power consumption of the *soft start* shown over time.**

sidered to be unusable. When reconnecting to the Thread network, there are two situations left. The first is a *soft start*, where the device is still marked as operational in the Thread network. The second option is the *reset start*, where the device needs to create a completely new connection with the Thread network. For both situations power measurements are made.

The first situation is the *soft start*, a single example of such a situation is drawn in Figure 5.5. The whole restarting and reconnecting to the network takes approximately 600ms. In the first half, one can see the boot sequence. In the second half two messages are send and responses received from the Router. The peaks from operating the radio of the microcontroller are clearly visible.

For the second situation, a complete reconnect, is also a single situation chosen and shown in the Figure 5.6. The total boot sequence in this situation takes approximately 7000 ms. During these sequences a lot of information is communicated back and forth between our Child device and the Router. The total amount of messages send and received by the Child is 21 in this first sequence. Sometimes a message is lost, however 21 messages are needed to send over all the information. Also device needs to send an acknowledgement packet out for every packet it receives.

5.2.3 Average Power Consumption per Situation

The data from previous subsection shown in the figures is quantified in Table 5.1. The table shows the average power consumption in each of the different scenarios.

Starting with the two different resets, the *soft reset* has on average a lower

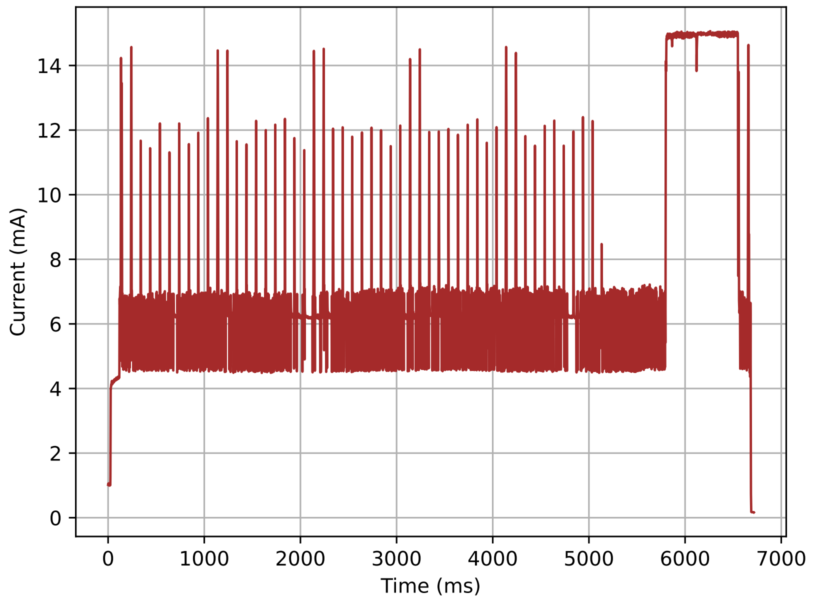

Figure 5.6: **Power consumption during a complete restart of the system until finishing the connection.**

power consumption compared to the *hard reset*. This was also visible in Figures 5.5 and 5.6, the higher and longer peaks of the *hard reset* do have a significant impact on the average power consumption. The *hard reset* takes longer to make the connection, this is due to the large amount of information needed to be shared before accepting the End Device into the network.

The Naive (A) implementation has high energy consumption during both the peaks period as well as during the off period. This was already clear from previous figures. The peaks power consumption of the Semi Naive and SSED (B & D) implementations are comparable, while the average of the SED (C) during peaks is lower than the other two. The Semi Naive implementation also consumes on average more power compared to the other implementations. Both SED and SSED (C & D) have comparable power consumption values during the off period. This is expected, since their implementation for the off period is the same.

An important note to take into account is the average time spent in each of the periods. The peaks period of the two Naive implementations (A & B), is shorter compared to the SED and SSED (C & D) implementations. The time it takes to read and write to and FRAM as well as the time to wait for the next time synchronisation point is included in the peak period of the SED and SSED (C & D), this is the main reason the peak periods take longer on the two devices (C & D). The power consumption of the devices during regular startup is not displayed, since this is included in the peaks of the previous section.

Table 5.2 is the total energy consumption of the *soft reset* and *hard reset*

Situation	Avg. Current (μA)	Avg. Time (ms)
During soft reset	2720.475	6070
During hard reset	8161.174	66906
Naive during peaks	9791.798	3.9
Naive during off	6120.223	4948.8
Semi naive during peaks	2903.496	11.9
Semi naive during off	44.844	5368.9
SED during peaks	2011.132	186.5
SED during off	7.893	4455.8
SSED during peaks	2740.544	164.7
SSED during off	8.585	4455.7

Table 5.1: **Average power consumption during two startups (soft and hard reset), as well as during the peaks and off period of the four different implementations (Naive (A), Semi Naive (B), SED (C) and SSED (D)).**

Startup type	Total Energy Consumption (mC)	Time (ms)
Soft	16.513	6070
Hard	546.31	66906

Table 5.2: **Energy consumption of the different booting sequences.**

startup is calculated. As expected the *soft startup* consumes a lot less energy for the entire startup compared to the *hard reset* startup.

5.3 Impact on Use Cases

In order to conclude which implementation consumes the least amount of energy during operation, a few more calculations need to be done. To perform the comparison we calculate the total power consumption of the device for an entire (theoretical) day (24 hours). If we take a look back at Section 4.6 we described the amount of times a device needs to turn on and send out data per day. With the measurements recovered from Section 5.2, the total amount of expected power consumption per situation can be calculated. From the results we take the average peak power consumption and the time it spends in such a peak. The peak power consumption is added to the total and the time spend in the peak is removed from our theoretical day. The rest of the time is calculated, during this time the device is considered to be in the low energy state T_{low}. As seen in the results do SED and SSED have multiple smaller peaks with a high current consumption in the middle, these implementation will therefore take more time per peak compared to the Naive and Semi Naive implementations. The entire stretch from start of the first to end of the last peak is considered to part of T_{peak}, where $T_{\text{total}} = T_{\text{peak}} + T_{\text{low}}$. Both timings are calculated for an entire day and shown in Table 5.3.

The power consumption per stage is shown in the same table. P_{peak} shows the power consumption during the peak or active time of the system. This value is obtained by utilizing the time mentioned above in combination with

Use Case	Implementation	T_{peak}(ms)	T_{low}(m)	P_{peak}(μC)	P_{low}(μC)	P_{total}(μC)
1. Thermostat	A. Naive	562	1439.99	5499	528764561	528770060
1. Thermostat	B. Semi Naive	1714	1439.97	4976	3874099	3879075
1. Thermostat	C. SED	26856	1439.55	54007	681743	735750
1. Thermostat	D. SSED	1423008	1416.29	3899042	729527	4628569
2. Valve Controller	A. Naive	33696	1439.44	329918	528561780	528891698
2. Valve Controller	B. Semi Naive	102816	1438.29	298475	3869566	4168041
2. Valve Controller	C. SED	1611360	1413.14	3240445	669237	3909682
2. Valve Controller	D. SSED	1423008	1416.29	3899042	729527	4628569
3. Flood Sensor	A. Naive	11232	1439.81	109973	528699260	528809233
3. Flood Sensor	B. Semi Naive	34272	1439.43	99492	3872639	3972131
3. Flood Sensor	C. SED	537120	1431.05	1080148	677716	1757864
3. Flood Sensor	D. SSED	1423008	1416.29	3899042	729527	4628569
4. Window Sensor	A. Naive	4	1440.00	39	528767976	528768015
4. Window Sensor	B. Semi Naive	12	1440.00	35	3874175	3874210
4. Window Sensor	C. SED	241	1440.00	485	681953	682439
4. Window Sensor	D. SSED	1423008	1416.29	3899042	729527	4628569

Table 5.3: **Time and power consumption of the different use cases in combination with the different implementations introduces in Section 4.6. With T_{peak} the time in peak power consumption, T_{low} the time in low power consumption, P_{peak} power consumed during peak period, P_{low} power consumed during low period and P_{total} total power consumed for this specific use case and implementation combination.**

the average power consumption during the peaks. In combination with the P_{low}, the power consumption during off or standby time, we can calculate the P_{total} power consumption, where $P_{\text{peak}} + P_{\text{low}} = P_{\text{total}}$. Here we see the core difference between regular, battery operated, devices and the battery-less implementations. The two naive implementations (A & B) both consume less energy during the peak period (P_{peak}) compared to the low period (P_{low}), while the battery-less implementations (C & D) both consume more energy during the low period in all of the use cases. Optimizations during operation become therefore becomes more important. As discussed in Chapter 5, the overhead incurred by creating and storing checkpoints into FRAM seem to have the most effect on these implementations. From the results, we also discovered the average power consumption during on time to be lower.

Important to note about the results in Table 5.3, is that the energy consumption of the SSED (D) implementation is for every use case the same. This is due to the maximal interval timer which can be set for such a device (10 seconds). This maximum interval value forces the device to communicate with the router node more often than required by our use case, thereby increasing the energy consumption significantly. This is also the case, to a lesser extent, for the SED (C) implementation. The maximum interval period for the SED to communicate is 18.6h, which is less than the needed interval period of the Window Sensor use case, set at 24h. Also for the SED is the power consumption increased with a factor $24/18.6 \approx 1.29$ in order to account for this difference.

Figure 5.7 depicts the total power consumption per use case and implementation as a visual representation of the values from the last column of Table 5.3. The power consumption scale is set to logarithmic in order to fit all the data in a readable manner. Only with the Valve Controller use case do the battery-free implementations (C & D) compare in total power consumption with the Semi Naive (B) implementation.

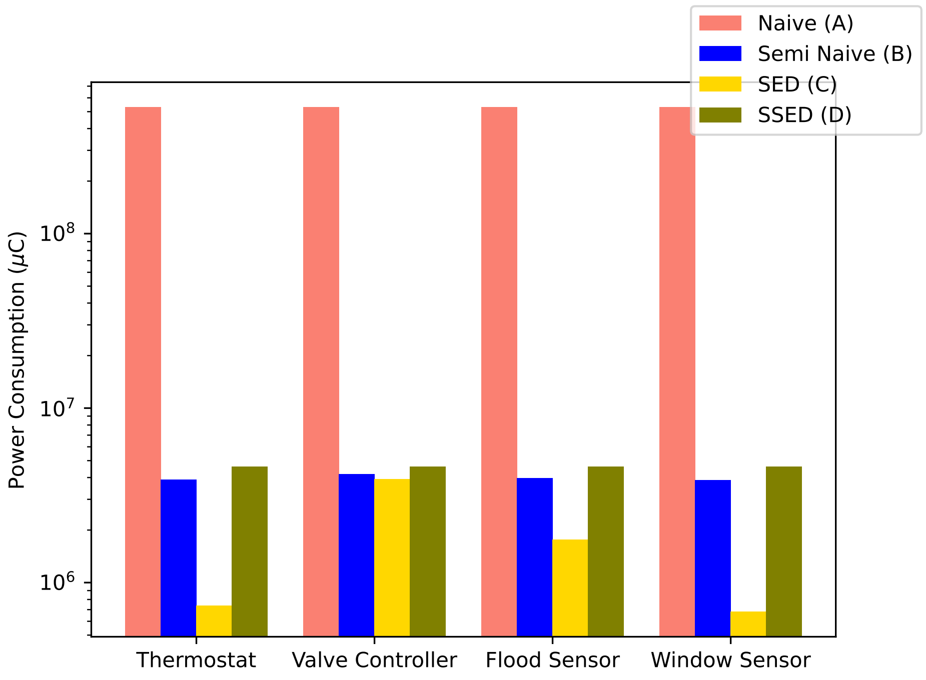

Figure 5.7: **Total power consumption of different implementations per use case.**

From these results we can conclude the SED (C) implementation to be the implementation consuming the least amount of energy. The implementation requires less power than any of the other implementations needed for the same use case. The next best solution is the Semi Naive (B) implementation, this implementation does not use the new intermittent hardware. However due to its short peak time, the total consumed power stays low. Next in power consumption is the SSED (D), the main reason for consuming more power, is the relatively short maximum interval time. The least performant implementation is the Naive (A) implementation, this implementation consumes in every case the most amount of energy.

Chapter 6

Conclusion

To conclude, we started with a brief introduction of battery-free devices in Section 1, including the origins, the challenges of the technology and setting the goal of this Thesis.

Followed by the current state of the research on battery-free devices in Section 2. We discussed different other relevant research topics for hardware about energy storage, energy generation, different ways of keeping track of time, storing information in non-volatile memory and discussed sensors and peripherals. For the software aspect of the battery-less paradigm, we discussed memory persistency, storing system and peripheral state, the ability to debug the system, scheduling tasks to be energy aware and most importantly for this research, other communication options for intermittently operating devices. In addition did we look at other adjacent technologies, transient computing and energy- or power-neutral systems.

Section 3 takes a look at the Thread protocol. The protocol is operating in the Network, Transport and Session layer of the OSI model. Utilizing the 6LowPAN, UDP and TCP technologies to transfer information around and maintaining connections, in combination with encryption and commissioning for protection and ease of use. Thread has different roles for devices, a Full Thread Device can be operating as a Router or End Device, the Router routing traffic to and from End Devices. A Minimal Thread Device can operate as a regular End Device or as a Sleepy End Device (SED). The Sleepy End Device can also take a special form as a Synchronised Sleepy End Device (SSED), checking for incoming packets without sending out a request. The topology of a Thread network is focused around a mesh of Routers, with the different End Devices directly connected to any one of the Routers.

Section 4 describes the implementation of the Thread protocol in the form of a library called OpenThread. This library consists of multiple layers. The Hardware Abstraction Layer, implemented by the hardware manufacturers. The Platform Abstraction Layer, which uses function calls in the Hardware Abstraction Layer to operate independently of the hardware platform used. The OpenThread Core Stack, where the operation is written as designed by the Thread protocol. And an Application Layer on top, using functions exposed from the OpenThread Core Stack to hook into the OpenThread library. By writing all the code needed to operate intermittently or battery-free in the outer layers of this stack, the OpenThread library and in extension the Thread protocol is unaware of the changes made. By utilizing the configuration of Sleepy End Devices and Synchronised Sleepy End Device, our implementations can run on minimal amounts of energy. The different implementations used in our tests are; the default implementation Naive (A), a default configuration with basic improvements called Semi Naive (B), a Sleepy End Device (C) capable of operating intermittently and a Synchronised Sleepy End Device (D) capable of operating intermittently. Even though these implementations are made, an implementation on the application layer is missing. Therefore four use cases are described with varying amount of traffic, which are used to determine the amount of energy consumption in these situations.

Section 5 starts of with the description of the test setup. The hardware is based on a design made in previous research, called a mote. This mote has the capability to run intermittently with an external RTC for timekeeping and FRAM to store checkpoints. A power profiler is used to determine the amount of energy consumption used during operation. The system is set up to go to sleep or turn off after the device is connected to the network and does not have an pending jobs to do. If it is possible to sleep or turn off, a checkpoint is made in case of the intermittently operating device, otherwise the device goes directly to sleep for the Semi Naive implementation B and do nothing for the Naive (A) implementation. The connection with the network is normally made by a commissioner, however to speed up the process, information about the network is already stored at the time of programming the device. The devices are all operating at 5 second interval, at the interval the device checks for possible messages pending at the Router node. The current consumption is measured during a time of 120 seconds.

The results are divided into two parts, the power consumption during peak periods and during low periods. For the Naive (A) implementation are both the peak and low power consumption the most compared to the others. The power consumption during peak is comparable for Semi Naive (B) and SSED (D) with the SED (C) consuming the least amount of power in this stage. However the time spend in peak period is longer for SED (C) and SSED (D), since they need to load and store the checkpoint as well as synchronising the clocks, which is not needed for the Semi Naive (B) implementation. The low power consumption of Semi Naive (B) is higher compared to SED (C) and SSED (D), which perform comparably during this period. In addition did we also measure the power consumption during soft and hard reset, where the hard reset is a situation where the Router does not remember the End Device during reconnecting. A soft reset is the moment when the End Device loses connection while the information about this device is still stored by the Router. In this situation the amount of energy required to restore the connection after a hard reset is a lot higher compared to the soft reset.

The impact of these results on the use cases show the SED (C) implementation out performing all the other implementation in every use case. The SSED did not perform as well, due to the high interval rate needed for this implementation. The Semi Naive (B) implementation consumes more power during the low periods, the SED (C) performs better when the interval between messages in longer.

6.1 Discussion

The results show a promising future of the intermittently operating devices in a Thread network. However, there are always improvements possible. See the list of improvements, with explanations below.

- Extra costs
- Semi Naive too simple
- Long boot timings
- Incomplete Thread Network
- Performance optimisations
- Test result accuracy

Extra costs. In this thesis we did not include the extra cost of hardware or development of intermittent devices. When designing and building devices the complexity increases with every system added. Introducing the RTC and FRAM to a hardware designs increases the overall cost per device as well as increases the amount of designing and testing of the hardware. In the next step, writing software for this hardware, the addition of more code does increase the cost as well. All these different costs should be taken into account when creating a marketable product.

Semi Naive too simple. The Semi Naive (B) implementation is too simple for a state of the art comparison. This implementation is a simple change from Naive implementation, however it is far from the the best optimized code possible. Due to a lack of strongly optimized code this is the best alternative for this research. To test the intermittent version to a state of the art implementation of an optimized implementation would be preferred. However, the current tested implementations do paint a worthy image of the usability of the OpenThread library.

Long boot timings. The booting sequence after a sleep period is still relatively long. The time for an SED or SSED (C & D) is 186.5 or 164.7 ms on average. As mentioned before and shown in Figure 5.4, the time between the system ready to operate after waking and actually sending a packet out takes a long time. If this time would be reduced, the total energy consumed for the implementations SED and SSED (C & D) could be further improved, since the peak time is the majority of power consumption of these devices.

Incomplete Thread network. Our implementation does not perform as a complete Thread network. The device is programmed to only send out packets for updating the timer at the Router. This uses the same underlying systems as sending out a custom packet to another node in the network, however this is never tested. Also, the End Device only receives acknowledgement packets which are not handled the same way as a regular UDP or TCP packet would be. This includes the ability to send and receive data from outside the Thread network, via a border router. This detail should be taken into account for the use cases as well, the current calculations only account for sending out a single packet. If the use case requires to receive a message, is this not accounted for in these calculations.

Performance optimisations. Further intermittent operation performance optimisations are not included. The intermittent operation itself could be further optimised to reduce the power consumption even further. The device could be configured to only receive a message at a specific time and store it on a specific place in FRAM without actually reading its content, at the next full boot, when the complete stack is loaded from FRAM again, the device will process the received packet. Also the amount of checkpointing can be reduced. If no new interesting data needs to be stored, the creation of the checkpoint can be skipped. In the current state the device creates a full checkpoint of the system, this can also be reduced to only include parts of the memory which do need storing.

Test result accuracy. The last point of attention is the accuracy of the results. As mentioned in the description of the test setup, the accuracy of the measurements are not particular accurate with $\pm 10\%$. Resulting in measurements which are hard This is due to the hardware used to measure the power consumption. Changing out this power profiler for a different tool could increase the accuracy of the measurements. However, the results are spaced far apart enough to conclude with confidence which device used less energy compared to the other devices.

Bibliography

[1] Fulvio Bambusi, Francesco Cerizzi, Yamin Lee, and Luca Mottola. The case for approximate intermittent computing. *CoRR*, abs/2111.10726, 2021.

[2] Carsten Bormann, Zach Shelby, Samita Chakrabarti, and Erik Nordmark. Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 6775, November 2012.

[3] Alexei Colin and Brandon Lucia. Chain: Tasks and channels for reliable intermittent programs. *SIGPLAN Not.*, 51(10):514–530, oct 2016.

[4] Alexei Colin, Emily Ruppel, and Brandon Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 767–781. ACM, 2018.

[5] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 53–67, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(3), sep 2020.

[7] Jasper de Winkel, Haozhe Tang, and Przemysław Pawełczak. Intermittently-powered bluetooth that works. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 287–301, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Wesley Eddy. Transmission Control Protocol (TCP). RFC 9293, August 2022.

[9] T. Eshita, T. Tamura, and Y. Arimoto. 14 - ferroelectric random access memory (fram) devices. In Yoshio Nishi, editor, *Advances in Non-volatile Memory and Storage Technology*, pages 434–454. Woodhead Publishing, 2014.

[10] Francesco Fraternali, Bharathan Balaji, Yuvraj Agarwal, Luca Benini, and Rajesh K. Gupta. Pible: Battery-free mote for perpetual indoor BLE applications, 2018.

[11] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd: A portable testbed for the batteryless iot. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 83–95, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 439–455. USENIX Association, April 2021.

[13] Kai Geissdoerfer and Marco Zimmerling. Learning to communicate effectively between battery-free devices. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 419–435, Renton, WA, April 2022. USENIX Association.

[14] Google. Openthread. `https://github.com/openthread/openthread`.

[15] Google. ot-nrf825xx. `https://github.com/openthread/ot-nrf528xx`.

[16] OpenThread Group. Link. `https://openthread.io/reference/group/api-link-link`.

[17] Thread Group. Thread specification. `https://www.threadgroup.org/ThreadSpec`.

[18] Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, page 330–331, New York, NY, USA, 2014. Association for Computing Machinery.

[19] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA, 2017. Association for Computing Machinery.

[21] IEEE. Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pages 1–800, 2020.

[22] Neal Jackson, Joshua Adkins, and Prabal Dutta. Reconsidering batteries in energy harvesting sensing. In *Proceedings of the 6th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, pages 14–18. ACM, 2018.

[23] Dhananjay Jagtap and Pat Pannuto. Reliable energy sources as a foundation for reliable intermittent systems. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, pages 22–28. ACM, 2020.

[24] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[25] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[26] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 129–144, Carlsbad, CA, October 2018. USENIX Association.

[27] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Discovering the hidden anomalies of intermittent computing. In *18th ACM International Conference on Embedded Wireless Systems and Networks (EWSN), Delft (The Netherlands), February 2021.*, 2021.

[28] Phillip Nadeau, Dina El-Damak, Dean Glettig, Yong Lin Kong, Stacy Mo, Cody Cleveland, Lucas Booth, Niclas Roxhed, Robert Langer, Anantha P. Chandrakasan, and Giovanni Traverso. Prolonged energy harvesting for ingestible devices. *Nature Biomedical Engineering*, 1(3):0022, 2017.

[29] Vishak Narayanan, Rohit Sahu, Jidong Sun, and Henry Duwe. Bobber a prototyping platform for batteryless intermittent accelerators. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, page 221–228, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Dr. Thomas Narten, Tatsuya Jinmei, and Dr. Susan Thomson. IPv6 Stateless Address Autoconfiguration. RFC 4862, September 2007.

[31] J. Postel. User Datagram Protocol. RFC 768, August 1980.

[32] J. Postel. Internet Control Message Protocol. RFC 792, September 1981.

[33] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V. Merrett, and Alex S. Weddell. Restop: Retaining external peripheral state in intermittently-powered sensor systems. *Sensors*, 18(1), 2018.

[34] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, 2008.

[35] SEGGER. J-link edu. `https://www.segger.com/products/debug-probes/j-link/models/j-link-edu/`.

[36] Nordic Semiconductor. nrf52840 dk. `https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK`.

[37] Noric Semiconductor. Power profiler kit ii. `https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2`.

[38] Thread Group. Thread 1.2 base features. `https://www.threadgroup.org/Portals/0/documents/support/Thread%201.2%20Base%20Features.pdf`, 2019. Last accessed: Sep. 16, 2022.

[39] Mbed TLS. Mbed tls. `https://github.com/Mbed-TLS/mbedtls`.

[40] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 41–53, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Eren Yildiz and Kasim Sinan Yildirim. Defragmenting energy storage in batteryless sensing devices. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, pages 36–42. ACM, 2020.

[42] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. Moo: A batteryless computational rfid and sensing platform. *University of Massachusetts Computer Science Technical Report UM-CS-2011-020*, 2011.