Green Access Control System

JONAS LUNDAHL & JOS ROSENQVIST
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Green Access Control System

Jonas Lundahl
dat15jlu@student.lu.se
Jos Rosenqvist
jo3812ro-s@student.lu.se

Axis Communications AB

Supervisor: William Tärneberg

Examiner: Maria Kihl

June 17, 2021

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The number of fixed-installation resource-constrained devices, such as Internet of Things devices, has been growing rapidly in the past years. One challenge of accommodating these large numbers of connected devices is the need to power them all. In this thesis, we investigate ways to reduce the power consumption of the door controller in an existing physical access control system, without compromising on system functionality or latency. Existing power management solutions of other resource-constrained devices were evaluated and suggested. Protocols and hardware components present in the system were researched, with focus on how power consumption and latencies can be reduced. Power measurement tests were performed on Linux power management systems `CPUFreq` and `CPUIdle` to evaluate their impact, as well as when suspending the CPU. Our results show that `CPUFreq` and `CPUIdle` are a simple way to reduce overall power consumption without compromising on system latency, and that over 25% of total power can be saved by suspending the CPU when the device is not in use. These results suggest that the greatest power savings are found when suspending the door controller CPU, and that system adjustments must be made to accommodate a suspended CPU. With this work, we hope that this type of device could adopt a battery-based power solution, reducing upfront installation costs.

Keywords— power consumption, physical access control, embedded systems, Internet of Things

Popular Science Summary

The number of smaller electronic devices such as mobile phones, sensors, smart devices, scanners and smart home appliances are increasing rapidly in our society and becoming more commonplace every year. It is important to make these devices more power efficient to reduce their collective energy costs and negative impacts on the environment. We have implemented ways to let a device save as much as 25% power by entering different low-power modes.

In this thesis, we have investigated a physical access control system, which is used to allow or deny access to people who want to enter a building or a room. More specifically, we have looked at the door controller of this system, which is responsible for inspecting the card credentials of the person requesting access and then telling a connected door or gate to unlock. The device as it currently exists cannot be powered by a battery for longer periods of time, but if its power consumption is reduced, it could be possible to make a battery-powered variant of the product which would be simpler and cheaper to install.

We have compared how effective different power saving features are. We found that the energy consumption can be reduced somewhat by enabling certain features in the door controller's operating system and can be reduced much more by forcing the device's processor into a sleep mode when nothing is happening. For an access control system, it is common for nothing to be happening, since there are many times of day when nobody is trying to enter or exit.

When the processor is in its sleep mode, it cannot do anything until something causes it to wake up. A large part of our work has been to make sure that the processor wakes up quickly and reliably when someone wants to enter or something else happens. This turned out to be more complicated for one of the two protocols the device uses to communicate with card readers, due to that protocol being more complex and requiring the device to regularly send messages to the card reader.

Acknowledgments

We would like to give a special thank you to the following people:

- Our main supervisors at Axis for their guidance and suggestions throughout the thesis and helping us get in touch with other employees at Axis:
  - **Marcus Johansson**, *Expert Engineer, New Business Access Control, Axis Communications, Lund*
  - **Johannes Jakobsson**, *Experienced Engineer, New Business Access Control, Axis Communications, Lund*
- Our examiner and supervisor from Lund University (LTH) for their feedback and recommendations for this thesis report:
  - **Maria Kihl**, *Professor, Electrical and Information Technology, Faculty of Engineering, Lund University*
  - **William Tärneberg**, *Postdoctoral fellow in Broadband Communications, Electrical and Information Technology, Faculty of Engineering, Lund University*
- The following Axis employees who took the time to sit down with us and help us proceed in our work (in no particular order):
  - **Marcus Andersson**, *Engineer, Fixed Cameras Electronics*
  - **Andreas Hansson**, *Experienced Engineer, NB Access Control QA*
  - **Henrik Fredriksson**, *Senior Engineer, New Business Access Control*
  - **Rickard Andersson**, *Senior Engineer, New Business Access Control*
  - **Mats Larsson**, *Senior Engineer, New Business AC Sol E&M*
  - **Karl Johnsinius**, *Engineering Manager, New Business E&M*

Preface

Jonas Lundahl is the primary author of the following parts of the thesis:

- Abstract
- Related work, Limitations, Scientific contributions, Disposition (in Introduction)
- `CPUFreq`, `CPUIdle`, Suspend-to-RAM, `systemd` (in Technology)
- CPU power domains, Possible power savings (in The existing system)
- Experiments
- Results
- Encryption in lower power systems, Creating a battery-powered system, Replacing Ethernet with Wi-Fi (in Discussion)
- Software (in Future Work)
- Figures and tables throughout the whole thesis

Jos Rosenqvist is the primary author of the following parts of the thesis:

- Project goal (in Introduction)
- Theory
- Wake on LAN (in Technology)
- A typical installation, Ethernet connectivity (in The existing system)
- Entering Suspend-to-RAM, Wake on LAN, Wake on MCU message without dropping the message, Wake on OSDP (in Implemented improvements)
- Outcome of the implemented improvements, The design of the OSDP protocol (in Discussion)
- Hardware (in Future work)

The following parts have had significant contributions from both authors:

- Popular science summary
- Background (in Introduction)

- UART, RS-485, OSDP, Wiegand (in Technology)
- System internals overview, CPU–MCU communication (in The existing system)
- `CPUFreq` & `CPUIdle`, Wake on MCU message (in Implemented improvements)
- Conclusions

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background

The number of devices with Internet Protocol (IP) connectivity has been growing rapidly in the past years. One challenge of maintaining these large numbers of connected devices is the need to power them all. The upfront installation cost of connecting a device to the power grid as part of a fixed installation can be prohibitive, and the alternative of using batteries leads to an ongoing cost and inconvenience when batteries need to be changed, in addition to the environmental cost of manufacturing batteries [1]. Reducing the power consumption of a device not only lowers operating costs when a device is connected to the power grid but also reduces the frequency at which batteries need to be changed, making the power consumption an important attribute of connected devices.

For resource-constrained devices, minimal power usage is of utmost importance. These Internet of Things (IoT) devices are made to run 24/7 in order to meet modern society demands. Alam et al. [2] define IoT as the usage of the Internet which bridges the gap between devices and services, which can be treated as an extension of pre-existing Internet services. This is made possible with sensors, microcontrollers and actuators present in the "things", allowing for ubiquitous global connectivity. IoT endeavors to provide a future where digital and physical entities can communicate seamlessly. It was estimated that over 50 billion IoT devices were online in 2020, up from just 9 billion devices in 2012 [2].

1.2 Project goal

The goal of this master's thesis is to investigate ways to reduce the power consumption of modern IoT devices, with a focus on physical access control systems (PACS). To accomplish this, we have examined and made modifications to a PACS developed by Axis Communications. The system comprises a network door controller hardware unit and the software running on it.

The main responsibility of the system is locking and unlocking a door in response to events such as a person swiping their access card in a card reader at the outside of the door or pressing a button at the inside of the door. The logic for deciding who is allowed to unlock the door resides on the device itself. The system is also capable of maintaining an IP connection to a server, to which it can

report events such as who is entering, and from which it can receive policy updates such as granting a new card access to the door, revoking access for a card already present in the local credentials database, or forcing the door into a specific state (e.g. unlocking all exits during a fire evacuation).

For this type of system, keeping the idle state power consumption as low as possible is key to minimizing the total power consumption, as the system spends much of its time idling. However, care must be taken to ensure that power consumption optimizations do not lead to a degraded user experience, such as longer latency before a door is unlocked.

In our work, we implement and evaluate a set of software changes. To evaluate the power savings and whether the user experience is impacted, for each change we measure the power consumption of the device when idle and when actively used, as well as the response time of the device when a user requests the door to be unlocked by swiping a card.

1.3 Related work

Previous work on the topic of minimizing the power consumption in resource-constrained devices, e.g. a network door controller, is presented in this section. Some areas of interest include developer tools focused on power usage, inexpensive communication methods suited for embedded systems, and machine learning.

1.3.1 Access deep sleep states using Céu

Céu is a structured, synchronous, and reactive programming language targeting resource-constrained embedded systems, which has been adopted in real-time systems such as those found in the avionics and automobiles industries. Software infrastructure has been proposed for this language which encompasses a power management runtime and support for interrupt service routines. This infrastructure allows applications written in Céu to take advantage of the deepest possible sleep modes available in resource-constrained embedded systems, without extra programming efforts [3].

Because this solution is dependent on the underlying code being written in a specific language, applying this to our solution is not feasible. It would be more suitable for smaller systems with smaller code bases, or systems developed from scratch.

1.3.2 Energy efficient data transfer in resource-constrained devices

The constrained application protocol (CoAP) is a simple web transfer protocol, which uses a non-connection method to reduce energy consumption in low-power networks containing constrained devices. To allow for easy integration, the protocol easily interfaces with HTTP[1] and has very low overhead. A main feature of

[1] HyperText Transfer Protocol.

CoAP include using UDP[2] instead of TCP[3], which does not require a three-way handshake before initiating a connection, saving system resources. CoAP is also compatible with DTLS[4] to allow for secure data transfers [4].

This protocol can be combined with the Open Platform Communication Unified Architecture (OPC UA), which is a data exchange standard used in industrial communication between machines and PCs, in the context of Industrial Internet of Things (IIoT). OPC UA's independence from the physical transport medium and transport protocols makes the standard flexible to implement and easily expandable to meet new demands, unlike other pure IoT data protocols [5]. Both client/server and publish/subscribe communication mechanisms are supported by the OPC UA specification, and can adopt several different communication methods such as HTTP and TCP. Wang et al. [6] showed that by using CoAP in place of HTTP, which otherwise has a high resource cost in resource constrained devices, the resource requirements of the these devices can be reduced and thereby lower power consumption during data transmission. This solution is referred to as a CoAP-based OPC UA transmission scheme, which inherits the core advantages of both the OPC UA and the CoAP protocol.

Although our work does not deal with industrial machines like OPC UA does, CoAP could prove useful in reducing overall power consumption of the door controller's network connectivity, see section 2.3.4.

1.3.3 Using machine learning to optimize an MCU

Machine learning is a set of tools and algorithms that automatically improve through experience, which provide a way of deriving meaning from large datasets [7]. Some applications of machine learning include extracting structured data from unstructured data, image and speech recognition, data classification, and function optimization [8].

Using machine learning, it has been shown that both the performance and energy efficiency can be significantly improved in resource-constrained systems. Raykov et al. [9] has implemented a non-parametric Bayesian machine learning algorithm on a high-performance microcontroller unit (MCU). The MCU is part of an IoT device alongside a battery and an analogue passive infrared sensor (PIR), used for measuring the number of people present in a room. With this model, both memory usage and execution times have been significantly reduced on the MCU, leading to great power savings. Tests of this implementation show that battery lifetime can be extended by a factor of 2.5, allowing the system to operate for 36 days using a 2200 mAh battery.

This system is similar to the network door controller examined in this thesis, as seen in section 2.3.2. Instead of a PIR sensor, our MCU operates with I/O devices and a CPU. The paper shows that machine learning can be used to increase energy efficiency in resource-constrained devices, and that it is worth investigating if this is possible in our system. Although we do not explore machine learning solutions

[2]User Datagram Protocol.
[3]Transmission Control Protocol.
[4]Datagram Transport Layer Security.

in this thesis, we acknowledge that this is an alternative path for reducing the power consumption of a system.

1.4 Limitations

The primary focus of this thesis is to create software solutions which result in overall reduced power consumption for the network door controller, which are to be presented as a proof of concept to Axis Communications. Solutions which involve introducing new hardware to the system or altering the existing hardware are not considered for implementation in the proof of concept, but they may be discussed.

1.5 Scientific contributions

The result of this thesis would allow Axis Communications to move closer towards developing battery-driven devices and products, which are currently not present in Axis' largest divisions such as network cameras and access control. We also hope that our work could lead to cheaper installation costs for future Axis access control systems, and provide a good basis for further studies into energy efficient systems in not just other Axis products, but also other resource-constrained devices of similar complexity levels.

1.6 Disposition

In chapter 2, we present technologies present in the network door controller, as well as some theory regarding power consumption. Chapter 3 shows what has been implemented in our proof of concept, how it has been done, and why. Our testing methodology used for measuring the overall power consumption and system latency is described in chapter 4, and the results of the performed tests are found in chapter 5. Chapter 6 discusses the effectiveness of our changes, as well as what kinds of changes are likely to be suitable for a production system. Finally, suggestions for future work are found in chapter 7, followed by our conclusions in chapter 8.

Chapter 2
Theory & Technology

This chapter presents existing theory and technology relevant to this thesis. Available strategies for power management are explained, as well as the components and protocols of the existing door controller system.

2.1 Theory

Decreasing the power consumption of computers has been an active research area for a few decades. While power consumption was not too much of a concern for early mainframe computers and desktop PCs, it became an important problem around the turn of the millennium, as mobile battery-powered devices became feasible to manufacture and the performance of desktop CPUs started to become limited by power dissipation constraints [10].

This section presents existing theory about power consumption which is applicable to various types of computers (desktop PCs, mobile devices, embedded systems, etc.). We start by describing what causes power to be consumed inside a computer and then discuss different strategies for reducing this power consumption.

2.1.1 CMOS power consumption

The total power P used by a digital CMOS circuit, for instance a processor, can be divided into the dynamic power P_{dynamic} (the power used when switching states) and the leakage power P_{leak} (the power continually used regardless of whether states are being switched). The dynamic power P_{dynamic} can further be divided into the power consumed by short circuits that briefly occur during state switching, P_{short}, and the power used for charging the capacitors of gates, P_{charge}. In full, we have:

$$P = P_{\text{charge}} + P_{\text{short}} + P_{\text{leak}} \tag{2.1}$$

$$P_{\text{charge}} = \alpha \cdot C \cdot f \cdot V^2 \tag{2.2}$$

$$P_{\text{short}} = I_{\text{short}} \cdot V \tag{2.3}$$

$$P_{\text{leak}} = I_{\text{leak}} \cdot V \tag{2.4}$$

where α represents what percentage of the system is switching states each clock cycle, C is the capacitance of the circuit, f is the clock frequency, V is the supply voltage, and P_{short} and P_{leak} are the short-circuit current and leakage current respectively [11, 12, 13].

The voltage needed for the stable operation of a circuit increases as the clock frequency increases [14]. Thus, increasing the clock frequency affects not only a linear factor of P_{charge} but also a quadratic factor, meaning the clock frequency can have a large effect on the power consumption of a device.

2.1.2 Dynamic voltage and frequency scaling

Many CPUs support switching between different clock frequencies and supply voltages at runtime, a capability referred to as dynamic voltage and frequency scaling (DVFS). The purpose is to allow systems to use a lower voltage and frequency and thereby consume less power at times when little work needs to be performed, while still allowing the use of high clock frequencies at times when there are greater performance requirements [15].

According to an analysis by De Vogeleer et al. [11], the curve for how much energy is used by a CPU when performing a given amount of work at different clock frequencies has a convex shape. In other words, for a given task, there exists a sweet spot frequency for which the amount of energy used is minimal. The inefficiency of a too low frequency is due to the execution time increasing more than linearly relative to the decrease in clock frequency, and the inefficiency of a too high frequency is due to the dynamic power consumption increasing quadratically with the increase in voltage.

2.1.3 Race to sleep

When analyzing the power consumption of a system, the dynamic power used by the CPU should not be viewed in isolation. A typical computer has many components which all contribute some level of baseline power consumption while powered on [16], including the CPU itself with its leakage power which increasingly stands for a large fraction of CPU power as the continued shrinking of transistor sizes is lowering the dynamic power more than the leakage power [14, 17]. A strategy for reducing the baseline power consumption of components is "race to sleep" – trying to finish all work quickly so that components then can be powered down or placed in a reduced power mode for as long as possible [16]. Making components transition to and from these low-power modes is however often associated with long latencies which may be undesirable depending on the application [18, 19].

The strategy of race to sleep stands in contrast to DVFS. Race to sleep saves more power the faster tasks finish, but DVFS attempts to save power in a way that makes tasks take longer to finish. How the two strategies should be balanced to achieve the lowest possible power consumption depends on various factors, such as how much power a specific system saves by entering low-power modes, and whether the program running on the CPU spends a lot of time waiting for memory access

(in which case reducing the CPU's clock frequency does not impact performance much). Modern hardware is trending towards DVFS saving less power than it once did [14].

2.1.4 Designing software for low power consumption

While supporting DVFS and low-power idle states in hardware and operating systems does reduce the power consumption of a device on its own, even greater reductions can be achieved by also adapting software applications to make better use of DVFS and idle states. The key idea is to reduce how much work needs to be done and how often the device needs to be woken up to perform work.

One example of an optimization that can be done is designing communication protocols in a way that let the device remain in a low-power mode most of the time when no messages are being transmitted to the device [20]. Another is to wait with performing work until there is a greater amount of work available, reducing the number of times the device must be woken up [21, 22].

2.2 Technology

Here, some power management subsystems and features found in the Linux kernel are presented, as well as the communication protocols used by the network door controller today.

2.2.1 `CPUFreq`

There exists a subsystem in the Linux kernel called `CPUFreq` (short for CPU Frequency scaling) which implements DVFS (see section 2.1.2), allowing the CPU to operate at multiple different clock frequencies and voltage configurations [23]. A clock frequency and its corresponding voltage configuration is referred to as an Operating Performance Point (P-state). For an integrated system or a system on a chip (SoC), it is often desirable to minimize the power usage by reducing the CPU clock frequency to a level that still allows the system to complete its tasks within some set time constraint. `CPUFreq` consists of three different layers of code, presented below:

- **The Core** | has common code infrastructure which defines the basic framework where other components of `CPUFreq` operate. It also implements the user space interfaces of `CPUFreq`.
- **Scaling Governors** | estimates the required CPU capacity for a given task, using parameterized scaling algorithms.
- **Scaling Drivers** | manage and access CPU P-states by talking to the hardware. These drivers provide the governors with information about the P-states supported by the hardware, as well as the ability for the governors to change CPU P-states on demand.

When the kernel is initialized, `CPUFreq` creates a directory `/sys/devices/system/cpu/cpufreq/` which contains `policyX` directories, where `X` is an integer starting

File name	Description
affected_cpus	list of online CPUs
cpuinfo_cur_freq	current CPU frequency (KHz)
cpuinfo_max_freq	maximum CPU frequency (KHz)
cpuinfo_min_freq	minimum CPU frequency (KHz)
cpuinfo_transition_latency	time to switch P-state (ns)
related_cpus	list of all CPUs
scaling_available_frequencies	list of possible CPU frequencies (KHz)
scaling_available_governors	list of possible scaling governors
scaling_cur_freq	current CPU frequency of all CPUs
scaling_driver	current scaling driver
scaling_governor	current scaling governor
scaling_max_freq	maximum CPU frequency of all CPUs (KHz)
scaling_min_freq	minimum CPU frequency of all CPUs (KHz)
scaling_setspeed	last frequency requested by governor (KHz)

Table 2.1: Attributes for a policy object in CPUFreq.

File name	Description
ignore_nice_load	count all processes towards CPU utilization (0 or 1). Processes marked as 'nice' are not considered if set to 1.
io_is_busy	count I/O activity towards CPU utilization (0 or 1).
powersave_bias	defines (in ‰) how much the CPU frequency should be reduced from the governor's initial target frequency.
sampling_down_factor	multiplier which affect how often decisions to adjust the clock frequency is made at high loads. Values greater than 1 imply making decisions less often during high loads.
sampling_rate	how often decisions are made to adjust the clock frequency (in μs).
up_threshold	what the CPU usage must be between samples to consider increasing the clock frequency.

Table 2.2: Attributes for a ondemand governor in CPUFreq.

from 0. A policy is a set of configurations for one or more CPUs in the system. The `policy` directory contains a number of extensionless files, summarized in Table 2.1. Changing the values of these attributes is done by overwriting their corresponding file with a new value. For example, the current scaling governor can be changed to `ondemand` using `echo ondemand > scaling_governor`. This governor creates another directory alongside the `policyX` directories, containing files which configure the `ondemand` governor, described in Table 2.2 [24].

2.2.2 CPUIdle

A CPU that enters an idle state suspends execution of the current program and instructions are no longer fetched from memory. `CPUIdle` belongs to the CPU idle time management subsystem in the Linux kernel which can put the CPU into an idle state [25]. A CPU may support multiple different idle states that save

different amounts of power, where the deeper idle states take longer to enter and exit; having higher entry and exit latencies than more shallow states [26]. Similar to `CPUFreq`, this subsystem has a governor that selects an idle state for the CPU to enter and a driver (which the governor invokes) that communicates to the actual hardware that it should enter the idle state.

A task is a sequence of instructions, code, or data that also contains context information which the CPU must load in before running the task's code. In the Linux kernel, a CPU is considered to be idle if it is running no tasks except for the special idle task. The special idle task becomes runnable if no other tasks assigned to the CPU are runnable. The idle task first calls the aforementioned governor code module to determine what to do, and then either invokes the driver to place the CPU in an idle state or runs more or less useless instructions in a loop, depending on whether the governor considers the power savings of an idle state to be worth the overheard of entering and exiting it.

2.2.3 Suspend-to-RAM

Suspend-to-RAM is a state that can offer significant power savings by putting the entire system into a low-power state, not including the memory [27]. Instead, the memory is put into a self-refresh mode which is used to retain its contents, due to its volatile nature. The state of all devices and the CPU itself, including system configuration and active files, is saved and stored in memory, which is retrieved once the CPU leaves its suspended state. Only a handful of events and devices can wake up the CPU once it has been suspended, such as a magic packet (see section 2.2.4). As an example, it is advisable for laptops and other mobile devices to enter a Suspend-to-RAM state when the device is running on batteries and the lid is closed or if the user has been inactive for a certain amount of time [28].

2.2.4 Wake on LAN

Wake on LAN (WoL) is a means of waking up a computer from a low-power or powered down state by sending data to it over a network [29].

Magic packet over Ethernet

The most common implementation of Wake on LAN involves sending a so-called *magic packet* over Ethernet. A magic packet is an Ethernet frame which at any point contains the following byte sequence: 6 `0xFF` bytes, followed by 16 repetitions of the 6-byte MAC address of the device to be woken up, for a total of 102 bytes. An example can be seen in Figure 2.1. A network interface controller (NIC) with magic packet support is capable of detecting magic packets in hardware and signaling the arrival of a magic packet to the rest of the system in some manner, for instance sending a signal to the power management circuitry, to allow the rest of the system to wake up when a magic packet is received. The system can thus listen for incoming magic packets without needing to power any major component of the system other than the NIC [29, 30].

The simplest type of magic packet is an Ethernet frame simply containing the 102-byte sequence as its payload. The destination MAC address specified in

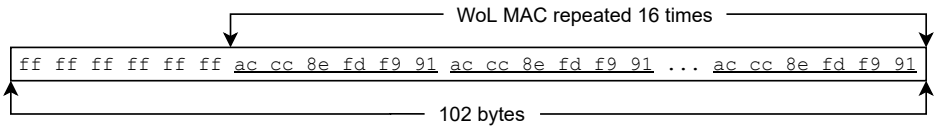


Figure 2.1: Magic packet payload example.

the Ethernet header can be either the unicast MAC address of the device to be woken up or the broadcast MAC address `ff:ff:ff:ff:ff:ff`, as long as the MAC address repeated 16 times in the payload is the unicast address of the device to be woken up [30]. This type of Ethernet frame is fully functional for waking up a device on the same network, but if Wake on LAN is to be used with a device on a different network, the 102-byte sequence must be wrapped inside a packet of a protocol on the network layer or higher. Typically UDP on top of IP is used for this purpose, with the most frequently used ports being UDP ports 7 and 9 [29]. Recall that since the NIC simply checks whether an Ethernet frame contains the 102-byte sequence anywhere inside it, it does not matter to the receiving device what protocols are in use on the network layer and above or what port number is used, as long as the magic packet is properly routed and received. The requirement imposed on the higher-level protocols is only that they do not compress, encrypt, split or otherwise alter the 102-byte sequence.

Sending a magic packet across networks

Sending a magic packet to a device on another network can however be troublesome in practice. The destination IP address can either be a unicast address or a subnet-directed broadcast address, and each approach brings different problems.

The problem with using a unicast address is that the router at the destination network may not be aware of the device which the magic packet is directed to. To any device on the network, a device waiting for a magic packet is for all intents and purposes powered off, as it does not send any data or acknowledge any received data. If the router has a mapping between the device's IP address and the device's MAC address in its cache from earlier, this is not a problem, and the router can correctly forward the magic packet as an Ethernet frame addressed to the device's MAC address. However, if the mapping is not in the cache, the router will fail in its attempt to use the Address Resolution Protocol (ARP) to find out the MAC address of the device, as the device does not respond to ARP request messages. The router then typically assumes that the device is not connected to the network and drops the packet [29, 30].

Using a subnet-directed broadcast address avoids the problem of ARP not functioning, as the receiving router can turn the packet into an Ethernet broadcast without needing to know anything about which devices are currently connected to the network. The magic packet then reaches every device on the network, and is ignored by all devices other than the one whose MAC address is specified in

the 102-byte sequence [30]. However, most routers actually entirely block subnet-directed broadcasts by default as they can be used for denial of service (DoS) attacks such as the Smurf attack [29]. This protection can usually be disabled, but this is inconvenient both in the sense that network administrators must spend time reconfiguring the network in order for Wake on LAN to work and in the sense that it lowers the security of the network.

One way of reliably waking up devices across networks is to place an always-on "wakeup server" on the target network that listens to incoming wakeup requests from other networks using some protocol and generates a magic packet on the local network in response. This sidesteps the aforementioned problems, as the actual magic packet is never sent across different networks. This does require additional hardware which in itself consumes some power, but a server that fulfills this role can be implemented using low-power embedded hardware [31, 32, 33].

Other variants of Wake on LAN

Some NICs support waking up not only on magic packets but on any network activity directed to the device, on network broadcast activity, and similar. Operating system facilities can be used to configure which types of networks activity should cause the system to wake up [34].

There is also a variation of Wake on LAN for Wi-Fi networks, known as Wake on Wireless LAN (WoWLAN). WoWLAN is less widely supported than WoL, and the power usage during WoWLAN standby is higher than for WoL because the Wi-Fi NIC needs to occasionally communicate with the network in order to retain the ability to decode packets if the network is encrypted [35]. It is possible to construct a mechanism that wakes the device when any Wi-Fi traffic is detected [36], without any need to periodically transmit data, but this is not especially useful in areas where Wi-Fi networks also are used by devices other than the device to be woken up since any network activity at all would trigger a wakeup.

2.2.5 UART

A Universal Asynchronous Receiver Transmitter (UART) is a device used for transmitting serial data. In UART communications, two or more processors have one UART device each which communicate with one another, using two wires and a set of sending and receiving (Tx and Rx) pins on both ends [37]. Due to the asynchronous nature of the communication, UARTs use special start and stop bits to know when to read the incoming bits, at a specific frequency defined by a pre-configured baud rate[1]. Both UART devices must operate at a similar baud rate (within 10% of one another) to avoid timing issues during data sampling [38]. UART devices send and receive data to and from the processor via a data bus, shown in Figure 2.2. The transmitting UART receives a byte from the data bus, creates a UART packet with the data bus byte as its data frame, sends it over the wire to the receiving UART, which unpacks the bus byte and sends it to the corresponding data bus on its end [37].

[1]The measured data transfer speed over the connection is known as the baud rate, expressed in bits per second.

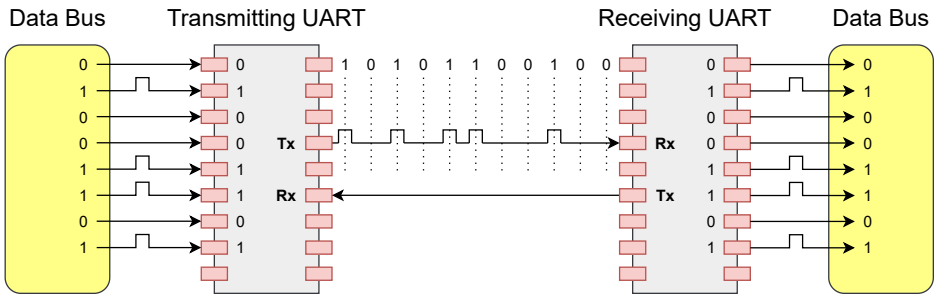


Figure 2.2: UART interaction with data bus and transmission example.

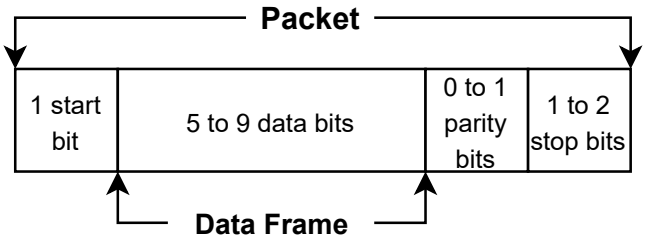


Figure 2.3: UART packet format.

The packet format used by UARTs can be seen in Figure 2.3. The data frame contains the actual data being transmitted, with a length of at most 9 bits. The parity bits are used for checking if the data has been inadvertently changed during transmission. The stop bits signal the end of a packet, and correspond to driving the data transmission line from a low voltage to a high voltage for at least two bit durations. It is up to the transmitting UART device to frame the data bits with the start bit, parity bit, and stop bits during transmission. The UART device on the receiving end detects these bits to determine the start and end of the actual data bits which are passed on to the receiving processor [37, 38].

2.2.6 RS-485

There are different electrical-level standards available for use with UARTs. One of them is RS-485, which is used in various consumer, medical, and industrial applications [39]. Although it is currently maintained by the Telecommunications Industry Association under the name TIA-485 [40], it is still commonly referred to as RS-485, the name it was originally released under by the Electronics Industries Association [41].

Some key features of RS-485 include allowing up to 32 devices[2] to be connected

[2]This limitation can be removed with the help of automatic repeaters and high-impedance drivers and receivers, allowing for thousands of additional nodes on the network [42].

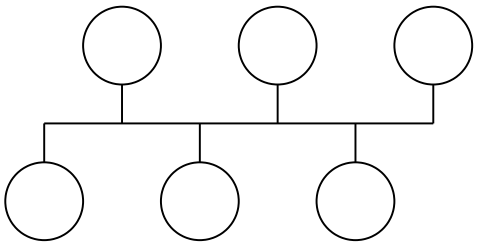


Figure 2.4: RS-485 daisy chain bus structure.

to the same bus and having a maximum cable length of up to 1200 meters. Shorter cable lengths allow for bandwidths up to 10 Mbps, whereas for much longer cable lengths, the bandwidth is limited to 100 kbps. It is suggested by the RS-485 standards to connect nodes in the network in a daisy chain, connecting the devices to the main cable trunk via short network stubs, see Figure 2.4. The bus can be full-duplex or half-duplex, requiring four or two wires respectively. Full-duplex allows a node to simultaneously send and receive data from the bus [39].

2.2.7 OSDP

Open Supervised Device Protocol (OSDP) is an open standard protocol for electronic access control systems, developed and owned by the Security Industry Association (SIA) [43]. OSDP is based on the RS-485 standard and is a bi-directional protocol, meaning that an access control unit (ACU) can both send and receive data to and from a peripheral device, e.g. a card reader [44]. Up to 127 different peripheral devices can be connected and addressed simultaneously, using standard RS-485 wiring up to 1200 meters (see section 2.2.6).

The structure and contents of OSDP messages can be seen in Tables 2.3a, 2.3b, and 2.3c [44]. `SOM` is used for message synchronization and marks the beginning of every OSDP message. `ADDR` denotes e.g. the card reader; the intended recipient of the message, with the value `0x7F` being reserved as a broadcast address. The `CMND/REPLY` field is used to differentiate between ACU-initiated and peripheral device-initiated messages. The purpose and meaning of the message is also defined by this field. The message control information described in Table 2.3b is used primarily for error recovery and message acknowledgments.

The Security Control Block summarized in Table 2.3c is an optional block which is present when using the Secure Channel feature added in OSDPv2. With Secure Channel, the entire message is authenticated using an AES-128-based message authentication code (MAC), and the data block of the message (but not the surrounding fields such as `ADDR` and `CMND/REPLY`) is encrypted using AES-128. Without Secure Channel, the entire message is unencrypted, and data integrity is verified using CRC-16 or a simple 8-bit checksum.

This communication channel between the ACU and its peripheral devices uses an interrogation-reply scheme, meaning that only the ACU may spontaneously send messages over the OSDP channel. In order for the ACU to be notified of new events from a peripheral, such as a card being swiped in a reader, it must period-

Size (B)	Name	Meaning	Value
1	SOM	Start of message	0x53
1	ADDR	Physical address of peripheral device	0x00 - 0x7F
1	LEN_LSB	Packet length least significant byte	Any
1	LEN_MSB	Packet length most significant byte	Any
1	CTRL	Message control information	See Table 2.3b
1	SEC_BLK_LEN	Length of security control block (optional)	Any
1	SEC_BLK_TYPE	Security block type (optional)	See Table 2.3c
1	SEC_BLK_DATA	Security block data (optional)	Based on type
1	CMND/REPLY	Command/Reply code	-
1	DATA	Data block (optional)	Based on CMND/REPLY
4	MAC	Present for secured messages	-
1	CKSUM/CRC_LSB	CRC-16 least significant byte, or 8-bit checksum	-
1	CRC_MSB	CRC-16 most significant byte (optional)	-

(a) The fields of an OSDP packet.

Bit	Mask	Name	Meaning
0-1	0x03	SQN	Sequence number, used to confirm message delivery and to recover from errors.
2	0x04	CHKSUM/CRC	Bit set: 16-bit CRC in the last 2 bytes of the message. Otherwise, 8-bit checksum contained in the final byte of the message.
3	0x08	SCB	Bit set: Security Control block present in message, otherwise not present.
4-7	0xF0	-	Set to zero.

(b) The structure of the message control information (CTRL) field of OSDP packets.

Size (B)	Name	Meaning	Value
1	SEC_BLK_LEN	Length of security control block	Any
1	SEC_BLK_TYPE	Security block type	-
n	SEC_BLK_DATA	Variable length data (optional)	Any

(c) The structure of the optional Security Control Block of OSDP packets.

Table 2.3: OSDP protocol summary.

Color	Function
Red	Power
Black	Common or Data Return
White	Data One (D1)
Green	Data Zero (D0)
Brown	LED Control

Table 2.4: Wiegand conductor definition.

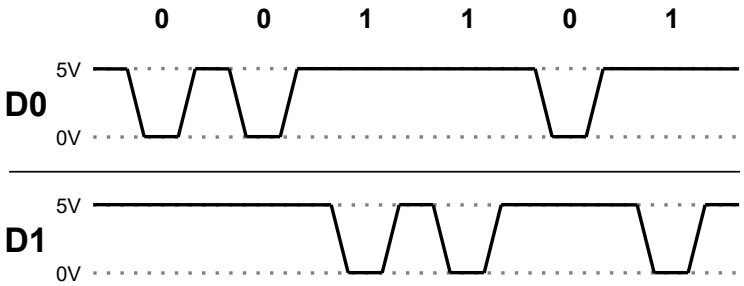


Figure 2.5: Wiegand bit sequence transmission example.

ically send poll request messages to the peripheral. The peripheral then responds to each poll request with information about whether anything has happened since the last poll request.

2.2.8 Wiegand

In the context of access control cards and readers, the term Wiegand is used to describe several different things. It may refer to a reader-to-card interface, binary reader-to-controller interface, an electronic signal carrying data, a 26-bit binary card data format, an electromagnetic effect, or a card technology [45]. For this thesis, we will focus on the reader-to-controller interface.

The physical reader-to-controller interface connects the reader and controller with a five conductor cable, defined in Table 2.4. Aside from the power and common ground lines, there are two data lines which the reader uses to send card data to the controller, and an LED control line which the controller can use to provide rudimentary information to the user (such as turning the reader's LED green when the door is unlocked) [46].

The two data lines D0 and D1 are operated as follows. When no data is being sent, both data lines are held high at 5 V. To send a bit of data, either D0 or D1 is held low for a moment, with D0 being held low for a binary zero and D1 being held low for a binary one [46]. An example of data transmission is shown in Figure 2.5, where the bit sequence `001101` is sent using the two data lines.

Under normal circumstances, at least one data line is held high at any given time. If the controller detects that neither line is high, it can assume that the connection to the reader is broken.

2.2.9 `systemd`

In a Linux-based system, `systemd` is a suite of tools that provides a system and service manager, which starts the rest of the system [47]. These tools also allow for other features such as aggressive parallelization capabilities, on-demand starting of daemons, and an dependency-based service control logic. `systemd` is meant to replace SystemV, which is an older, similiar system that has been used since the original Unix distributions that has less features [48]. Startup processes in `systemd` are managed via `.service` files, which are controlled by `.timer` unit files [49]. Both monotonic and realtime timers are supported, i.e. time relative to system startup and Earth time, respectively.

2.3 The existing system

This section presents the network door controller that we aim to make more energy efficient with our work.

2.3.1 A typical installation

The basis of a door controller installation is a *secured area* which is intended to be accessible only to authorized people, and a lockable door through which these people can enter and exit the secured area. The door controller is installed within the secured area, typically close to the door, and is responsible for unlocking and locking the door as needed.

Aside from being connected to an electronic door lock, the door controller is typically connected to a card reader at the unsecured side of the door (for those who wish to enter) and a push button and/or second card reader at the secured side of the door (for those who wish to exit). The access cards used with card readers may employ different technologies, for instance magnetic stripes or radio frequency identification (RFID). For the purpose of this thesis, the exact technology used by access cards is not relevant, and we usually refer to the action of a person presenting their access card to a card reader as "swiping a card" to keep the wording simple, even though this action may involve for instance a tap instead of a swipe depending on the technology in use.

The door controller used for our work supports communicating with card readers over either OSDP (see section 2.2.7) or Wiegand (see section 2.2.8). It also contains a number of simple inputs and outputs for connecting peripheral devices. Examples include door monitors (sensors which detect whether a door is open), the aforementioned electronic door locks, glass break detectors, fire detectors, and arbitrary analog or digital signals that the system administrator would like to monitor. The door controller is capable of providing power to peripheral devices as well, up to a limit.

2.3.2 System internals overview

A system component overview of the door controller in our work can be seen in Figure 2.6. The two primary components of this door controller are a central

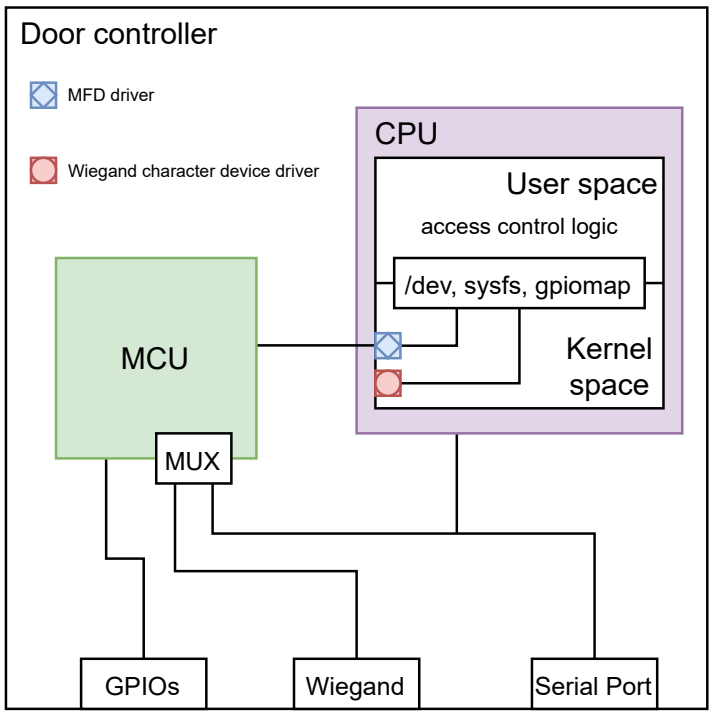


Figure 2.6: Hardware door controller system overview.

processing unit (CPU) and a microcontroller unit (MCU). The CPU runs a custom Linux distribution developed by Axis, and the MCU runs FreeRTOS, a real-time operating system for microcontrollers which consumes very little power [50].

An MCU is a complete computer system implemented on a single integrated circuit, often including additional modules such as timers and analog-to-digital converters [51]. The responsibilities of the MCU in this system are to communicate with card readers using the Wiegand protocol and to manage general purpose inputs and outputs (GPIOs) used for communicating with other peripheral devices. Compared to the system's CPU, the MCU has less processing power, but it is also responsible for fewer tasks, and can therefore poll inputs at regular intervals and react to any changes with very low latency. If this were to be done on the CPU instead, the polling would not be as regular, since the CPU has many other tasks to run and is not running an operating system with real-time guarantees. It would also be less energy efficient, since the CPU consumes more power than the MCU, and would use up CPU time which could have been spent on other tasks.

Unlike Wiegand, OSDP is handled entirely by the CPU without any involvement from the MCU. The task of regularly polling the RS-485 channel (not to be confused with the task of sending OSDP poll request messages, as described in section 2.2.7) is performed by a UART connected to the CPU (see section 2.2.5), freeing the CPU from needing to spend time handling RS-485 except when actual data is received or transmitted. The role of the CPU's RS-485 UART in the OSDP

case is similar to the role of the MCU in the Wiegand case, with the key difference being that the UART is implemented in fixed-function hardware.

2.3.3 CPU–MCU communication

The access control logic – the central decision-making code for when the door should be unlocked and which cards are allowed to unlock the door – runs in user space processes on the CPU. Since most peripheral device I/O is handled by the MCU, the CPU has to coordinate with the MCU in order for the access control logic to function. This is accomplished by the two communicating using a UART connection and a custom protocol.

The basis of the CPU–MCU protocol is the *message*. At any time, either the CPU or the MCU can decide to send a *command* message to the other party. Such messages can be referred to as either CPU-initiated or MCU-initiated. The second type of message is the *reply* message. The MCU sends a reply message whenever it receives a CPU-initiated message, but the CPU does not do the same for MCU-initiated messages. The different types of replies are ACK (the command was successfully completed), NAK (the command was invalid), and BUSY (the MCU currently cannot complete the command but may be able to do so later).

A message contains a type value, a session ID, and a variable number of data bytes. The type value indicates which type of command or reply the message is. The session ID is intended to distinguish a message from other messages, and is in particular used for determining which reply corresponds to which command. For CPU-initiated messages, the session ID is based on an incrementing counter which rolls back to 0 when it hits 0x3F. For MCU-initiated messages, the session ID is always 0x40. For replies, the session ID is copied from the command which is being replied to.

The CPU's end of the CPU–MCU protocol is implemented in a multi-functional device (MFD) driver in the Linux kernel. It exposes I/O devices to user space, allowing the access control logic to be alerted to events like a button press or the door opening. Similarly, the MFD driver allows the access control logic to send commands to the MCU, for instance in order to change one of the system's outputs. A Wiegand character device driver also exists, through which user space processes can receive card data from a Wiegand card reader when a card is swiped.

2.3.4 Ethernet connectivity

While the MCU is what handles most communication with peripheral devices, there is one form of communication which is handled exclusively by the CPU – the Ethernet connection.

An Apache HTTP web server runs on the CPU to provide a web-based configuration interface for system administrators. Multiple accounts can be created with varying permissions, and HTTPS can be enabled if an administrator provides a certificate for the device. The web interface can be used both when performing the initial setup of the system, with options such as setting the date and time and configuring how to communicate with connected peripheral devices, and also to

verify that the system continues to operate correctly, with options such as checking the state of the system and manually triggering locks and other devices.

The system can also communicate with an access control server. One purpose of this is to let the door controller automatically receive policy updates. For instance, if a new employee is to be granted access to all doors in a certain company building, the server can send the card data stored on the new employee's access card to the door controllers in that building. The server can also be made aware of events which occur at door controllers. For instance, if a door controller detects that its door has been opened despite the door controller commanding the lock to be in the locked position, the server can be alerted of this and perform an appropriate action, such as alerting security personnel.

2.3.5 CPU power domains

A power domain is a group of blocks or subblocks which is powered by power sources controlled by the same power controller. Power domains can be split into multiple subdomains, such as logic and memory subdomains. Every subdomain contains two entities called memory arrays and memory interface logic. The former is powered by a dedicated voltage rail, which enables the CPU's volatile memory to retain its information. The latter is powered by the same voltage source as the logic subdomain of the power domain.

In the network door controller CPU, there are multiple power domains, with the primary domains being the following:

- **Arm** | contains the Arm Core platform, except for memory arrays and interface logic.
- **Arm Memory array** | all memory arrays are connected to a separate and dedicated power domain.
- **Display domain** | is supplied by an internal regulator and contains for example the general interrupt service (GIS) and PCI Express (PCIe).
- **MEGA domain** | contains multiple items, such as channel 1 of the universal asynchronous receiver/transmitter (UART1) and on-chip random access memory (OCRAM).
- **SNVS/RTC low power domain** | contains only a counter comparator and compared data of the on-chip real-time clock (RTC), and is supplied by either an external battery or an external pre-regulated power supply.
- **Analog domain** | contains the phase lock loops (PLLs), low dropouts (LDOs), and the USB physical driver. To allow for continuous clocks during voltage scaling techniques, the domain is supplied with constant power.
- **Main SoC logic** | contains the rest of the logic of the SoC such as general-purpose I/O (GPIO), I2C, and UART channels 2 to 6. The domain is powered by an internal regulator.

2.3.6 Existing power measurements

In a power measurement document published by the developers of the network door controller's SoC, the chip where the device's CPU is located, multiple different use cases and the corresponding power consumption are presented. Some of the benchmarks in this document are presented below:

- In deep sleep mode, or Suspend-to-RAM (see section 2.2.3) as referred to in the Linux BSP and this thesis, there was a power consumption of approximately 24 mW.
- When the system was in idle, they recorded a total power usage of 34 mW.
- In an MP3 audio playback test, the power usage was 204 mW.
- During the Dhrystone benchmark, which is a synthetic benchmark, the system pulled 804 mW.

Chapter 3
Implemented Improvements

To reduce the power consumption of the network door controller, a number of software changes have been implemented. This chapter describes these changes, including how they were implemented and the reasoning behind our choices.

3.1 CPUFreq & CPUIdle

To begin with, we enabled two features which were already present in the Linux kernel but had not been enabled for the network door controller: `CPUFreq` (see section 2.2.1) and `CPUIdle` (see section 2.2.2), which enable DVFS (see section 2.1.2) and dynamically switching to idle states respectively. Since enabling these features is relatively simple, even if the resulting reduction in power consumption is small, enabling them is likely to be worth the effort as long as they do not cause any noticeable performance degradation.

In order to enable the `CPUFreq` and `CPUIdle` subsystems, we added the following lines to the configuration used when building Linux for the network door controller:

```
CONFIG_CPU_FREQ_DEFAULT_GOV_ONDEMAND=y
CONFIG_ARM_PSCI=y
CONFIG_CPU_IDLE=y
```

The first setting changes the default `CPUFreq` governor from `performance` (which always uses the maximum CPU clock frequency) to `ondemand` (which varies the CPU clock frequency depending on the CPU load). The second setting enables the operating system to communicate with system firmware that implements features related to power management [52], which on this system is required in order for `CPUIdle` to function. Finally, the third setting enables `CPUIdle`, making the kernel automatically transition the CPU to lower-power states when idle.

While it is possible to change the `CPUFreq` governor at runtime as described in section 2.2.1, changing the default governor in the compile-time configuration is more convenient long-term since the system resets to using the default governor after a reboot.

3.2 Suspending the CPU

The majority of the implementation work has in some way been related to placing the system's CPU in Suspend-to-RAM, the low-power mode described in section 2.2.3. Suspend-to-RAM provides greater power savings than `CPUIdle`, but must be explicitly entered and exited, which can be cumbersome. If the system is to make use of Suspend-to-RAM while functioning as normal from the user's perspective, it is important that the system wakes up from Suspend-to-RAM whenever something happens that the system needs to respond to. We have explored two different wakeup sources which we believe together cover the normally occurring types of interactions with the system. For the case where the access control server or system administrator wants to reach the network door controller over the network, we have made the system use Wake on LAN, and for the case where a peripheral device such as a card reader is providing new data, we have made the CPU wake up when a message is sent from the MCU.

3.2.1 Entering Suspend-to-RAM

The system can be made to enter Suspend-to-RAM using the following command:

```
echo mem > /sys/power/state
```

Initially, trying to run this command would fail with a "busy" error code. This turned out to be because of a bug in the handling of Wake on LAN in the Linux kernel, which we identified and fixed. The bug is described below.

Suspend failure related to Wake on LAN

Wake on LAN can be enabled and disabled from user space using the `ethtool` command. When this happens, the user's request is passed to the Ethernet driver, and it is up to the Ethernet driver to both enable or disable the Ethernet controller's support for Wake on LAN and pass on the request to the PHY (physical layer) driver if necessary. In our case, the PHY driver used on the system contained specific support for the PHY chip's Wake on LAN capabilities and enabled this support by default, but the Ethernet driver did not contain any code for enabling or disabling Wake on LAN on the PHY level.

Two pieces of code which run when suspending the system are directly relevant to the cause of the problem. The first is a check performed when suspending a PHY device which returns the busy error code if that PHY device has Wake on LAN enabled [53], the intent being that a PHY device must not be allowed to suspend if it is responsible for waking the system from suspend. The second is a piece of code which makes the suspend process skip suspending a PHY device in the first place if its attached network device (in our case the Ethernet controller) has Wake on LAN enabled [54].

The problem would occur when the user had not enabled Wake on LAN, resulting in Wake on LAN being disabled in the Ethernet driver but being left enabled in the PHY driver. The system would attempt to suspend the PHY device but then bail out in the belief that keeping the PHY device active was

necessary for allowing the system to wake up again, even though this was not actually the case. We fixed the problem by making the Ethernet driver update the PHY driver's Wake on LAN settings during driver initialization and whenever the Ethernet device's Wake on LAN settings change, so that Wake on LAN always is disabled for the PHY device when it is disabled for the Ethernet device.

3.2.2 Wake on LAN

Aside from the problem described above where having Wake on LAN disabled would prevent the CPU from suspending, Wake on LAN was functional on the network door controller before we started our work, albeit disabled by default. Wake on LAN can be enabled by a privileged user with the command `ethtool -s eth0 wol g`. Seeing as we were already modifying the system's Ethernet driver anyway, we made a small modification to the driver to enable Wake on LAN by default for convenience.

The `g` mode which is set by the `ethtool -s eth0 wol g` command makes the system wake up whenever a magic packet is received [34], as described in section 2.2.4. Unfortunately, the system's hardware only supports this mode, not modes such as waking up on any packet addressed to the device or waking up on any Ethernet activity. Only waking up on magic packets does prevent the system from spuriously waking up from unimportant network traffic, but it also means that anyone who wishes to communicate with the system over the network must explicitly send a magic packet first.

3.2.3 Wake on MCU message

With Wake on LAN covering waking up on network activity, what remains to be covered is waking up on peripheral device activity. The relevant peripheral device events can be categorized into three types:

- The user swiping a card in a card reader which is connected using Wiegand
- The user swiping a card in a card reader which is connected using OSDP
- A change in voltage of the signal from a device like a door monitor

In the existing system, the first and last of these are handled by the MCU constantly listening for changes and then sending a message to the CPU once something does happen. Using incoming messages from the MCU as a wakeup source for the CPU is thus a natural way to wake up on these two types of events. OSDP card swipes remain uncovered – how to handle them is discussed later in section 3.2.5.

Since the CPU uses a UART for communicating with the MCU, making the CPU wake up when the MCU sends a message to it is equivalent to making the CPU wake up when there is incoming data on the MCU-connected UART. Setting this UART (but not any of the CPU's other UARTs) as a wakeup source can be done using the following command:

```
echo enabled > /sys/devices/soc0/soc/2100000.aips-bus/
               21e8000.serial/tty/ttymxc1/power/wakeup
```

```
-----[wake-on-uart.timer]------------[wake-on-uart.service]---------
| [Unit]                       | [Unit]                             |
| Description=/*...*/          | Description=/*...*/                |
|                              |                                    |
| [Timer]                      | [Service]                          |
| OnBootSec=10s                | Type=oneshot                       |
| AccuracySec=1ms              | ExecStart=/usr/sbin/wake-on-uart.sh |
| Unit=wake-on-uart.service    |                                    |
|                              |                                    |
| [Install]                    |                                    |
| WantedBy=timers.target       |                                    |
-------------------------------------------------------------------
```

Figure 3.1: Wake on UART implementation. The description fields have been omitted due to space constraints.

We have placed this command in a bash script which runs every time the device boots up. This was accomplished by using timers and services in `systemd` (see section 2.2.9). The implemented timer and service are shown in Figure 3.1. The timer is configured to set to run the service ten seconds after system boot. The `[Install]` directive ensures that the timer is enabled during system startup. The bash script is installed inside the `/usr/sbin` directory on the network door controller, alongside other pre-existing scripts utilized by the system.

3.2.4 Wake on MCU message without dropping the message

While setting the MCU-connected UART as a wakeup source does make the CPU wake up on Wiegand card swipes and changed signal levels, the MCU message which causes the CPU to wake up is never actually received by the CPU. We believe that this is because the low-power mode the UART is in before the system wakes up is not functional enough to decode the received data, only to detect that there is some kind of incoming data. With this behavior, the user's first card swipe causes the system to wake up but not unlock the door, and a second card swipe would be required for the system to unlock the door. This is clearly not desirable.

To ensure that the suspended CPU is notified of an event without requiring the user to re-trigger the event, the MCU must first trigger a wakeup of the CPU, then wait for the CPU to wake up, and finally send the message describing the event. However, in the case where the CPU is not suspended, the event message should be sent as soon as possible so that the latency is kept low. This leads to a question: Should the MCU keep track of whether the CPU is suspended so that it can decide between two possible courses of action, or could the MCU always start out with the same course of action and then dynamically adapt depending on how the CPU responds?

It would be possible for the CPU to send a message to the MCU when it changes state to suspended or running. The MCU would then be able to keep track of the CPU's state by storing the last received state in memory. However, we chose

to not go down this route due to the potential of race conditions. Since the CPU by necessity cannot send a message to the MCU at the exact time as the CPU's UART switches between being able to receive messages and acting as a wakeup source, the MCU's view of the CPU's state will be incorrect for a short time around when the CPU suspends or wakes up. This means that the CPU would have to be able to handle the MCU communicating with it as if it is suspended when it in fact is just about to suspend or is in the process of waking up, which would involve some rather intricate kernel-level code modifications. Additionally, it would require the UART to instantly switch between being able to receive messages and acting as a wakeup source, with no period of time in between where incoming data is entirely ignored, and we do not know whether this is supported.

We are then back to the problem of how to make the MCU handle both a suspended CPU and a running CPU without knowing in advance which one it is dealing with. To start off, in the case where the CPU is running, it is ideal for the MCU to send the event message as its first course of action. This fortunately does not clash with the case where the CPU is suspended, as sending any data to the CPU's UART will result in a wakeup, no matter the contents. What remains is now how the MCU is to figure out whether the event message needs to be retransmitted, and if so when. We have identified two options for this:

- Making the CPU send a "resume" message to the MCU when it has woken up, as a hint to the MCU that it should resend any recently sent event message
- Making the CPU reply with an ACK message when it successfully receives an MCU-initiated message, and making the MCU retransmit sent event message periodically until it receives an ACK

The first option has the disadvantage that when the MCU sends an event message and does not receive a resume message as a response, it does not know whether this is because the CPU correctly received the response or because the CPU is still in the process of waking up and will send a resume message soon. Due to this, the MCU would need to use some kind of heuristic for deciding which event messages to retransmit when a resume message is received, for instance based on how recent the events are. This would not be more reliable than the aforementioned solution of the CPU sending a message to the MCU when it is entering suspend mode.

The second option has the disadvantage that the retransmission of an event message may happen a while after the CPU wakes up, causing the CPU's processing of the event to be unnecessarily delayed. If the MCU retransmits an event message every x ms for as long as no ACK reply is received, and the CPU takes y ms to wake up, the avoidable delay is $x - y$ ms, assuming that $y < x$. One way to decrease the delay is to decrease the value of x, but since y is not necessarily constant, setting x too close to y risks causing an avoidable delay of $2x - y$ ms due to the CPU not waking up in time to receive the first retransmission.

We decided to implement both of these options combined, avoiding the mentioned disadvantages of the individual options. Making the CPU send ACK replies to MCU-initiated messages was implemented in a similar way to the existing mechanism of the MCU sending ACK replies to CPU-initiated messages, with the roles

reversed. This included making the session ID for MCU-initiated messages range from 0x40 to 0x7F instead of always being 0x40. To keep the MCU-side implementation simple, we blocked the MCU from transmitting new messages until the most recently transmitted message is acknowledged by the CPU.

3.2.5 Wake on OSDP

In the existing system, card readers which use OSDP (see section 2.2.7) communicate with the door controller's CPU not through the MCU but directly with one of the CPU's UARTs. If the card reader were to send data to the CPU whenever a card is swiped, making the CPU wake up on OSDP card swipes could be accomplished by setting the UART as a wakeup source, analogously to what was done in section 3.2.3 for a different UART. However, this is not how OSDP works. The card reader can only send data to the CPU as a response to a poll request from the CPU.

Since the CPU cannot send poll requests when suspended, the MCU would have to be responsible for sending OSDP poll requests when the CPU is suspended (or alternatively, always). If the MCU then also listens to replies from the card reader, and forwards all replies that indicate that something has happened by sending a message to the CPU, our previous work described in sections 3.2.3 and 3.2.4 would allow the CPU to wake up on not only Wiegand card swipes but also OSDP card swipes.

We believe the MCU is capable of handling OSDP, but the Secure Channel feature in OSDPv2 poses a particular hurdle, as it requires the use of AES-128. The MCU in this system does not have hardware support for AES-128. However, earlier studies have shown that AES-128 can be implemented in software on memory-constrained devices comparable to this MCU, with performance which is acceptable for the small amounts of data typically sent over the OSDP protocol [55, 56].

We attempted to implement OSDP handling on the MCU, but unfortunately did not succeed in establishing RS-485 communication with the card reader, a required foundation for implementing OSDP. Eventually we had to abandon our attempts due to time constraints.

Chapter 4
Experiments

This section presents what experiments have been done to evaluate our implemented improvements, why they were done, and how they have been done. The results of these experiments are shown in section 5.1.

4.1 Measuring power usage

The test environment for measuring power usage is shown in Figure 4.1. Dashed lines correspond to Ethernet cables. A regular desktop computer is connected to a network switch supporting Power over Ethernet (PoE), alongside a network door station (used for simulating card swipes over Wiegand and OSDP), and the network door controller whose power usage is measured. To measure the current, an injector card is connected between the switch and the controller. This card is powered by an external power supply. This power supply is then connected to the 3A input of the multimeter, which is then connected to the injector card using the grounded LO input on the multimeter. With this setup, the current, measured in ampere (A), is recorded on the multimeter. The power W (measured in Watts) can be obtained using the elementary equation 4.1:

$$W = V \cdot A \tag{4.1}$$

where V is the power supply voltage of 48 V, and A is the recorded current of the multimeter. The following equipment was used during testing:

- CPX400DP Dual 420 watt DC Power Supply PowerFlex
- Keysight 34465A Digit Multimeter
- NETGEAR GS110TP — 10-Port Gigabit Ethernet Smart (PoE) Switch
- Network door controller by Axis
- Network door station by Axis

This multimeter has the feature of sampling the current up to 120 times per minute and saving the sampled data to a file, allowing us to perform more detailed analysis of the measurements post-testing, such as computing averages and standard deviations. The readings are stored in CSV (comma-separated values) format and retrieved from the multimeter with a USB stick.

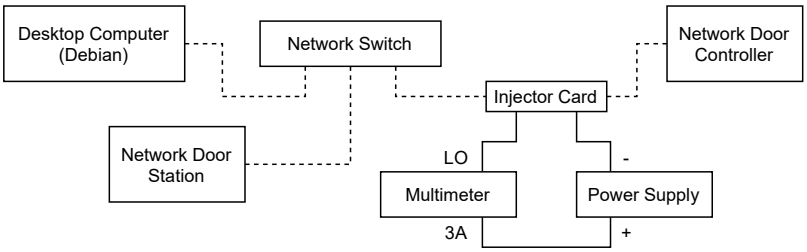


Figure 4.1: Test environment for power usage measurements.

4.2 Simulating card swipes

For simulating a system load, Apache JMeter [57] was used, which allows for sending API requests to both the network door controller and the network door station using HTTP or HTTPS. A pre-existing test plan was used in JMeter which contain tasks that are run on the two network devices in question, beginning with configuring the devices for the test environment and then sending card swipe events in JavaScript Object Notation (JSON) format. The network door controller can be artificially made to run its card authentication process by sending an HTTP POST request containing an authentication command alongside the relevant raw card data. Alternatively, a command can be sent to the network door station instead, which sends the supplied card data to the network door controller using Wiegand or OSDP like a normal card reader would (refer to section 2.3). For our tests, simulating card swipes was done by sending commands to the network door station, as it was deemed to be the most realistic test scenario. Invoking the authentication command directly on the network door controller would skip the initial steps of the process between card swipe and door unlocking, which was concluded to be undesirable since we want to analyze the overall latency of the system as a normal user would experience it.

JMeter was configured to command the network door station to send a card swipe event to the network door controller every 10 seconds. Tests were conducted with both with the system configured to use OSDP and with the system configured to use Wiegand, in case the system's power consumption and latency differs between the two. For the purpose of analyzing the system's latency in reacting to card swipes, timestamps for every step of the authentication process were collected from system logs available on the network door controller and summarized in a file using a pre-existing Python script. For our latency testing, we decided to perform approximately 30 simulated card swipes for each test run. The milestones, i.e. the points during the authentication process which the timestamps were recorded at, are explained in Table 4.1.

The steps in between the milestones are described in Figure 4.2. The arrows with solid lines denote the direction which events are sent, whereas dashed lines denote commands. Boxes with regular text are services available in the system and boxes with underlined text denote the events described in Table 4.1. In general, events are sent in the opposite direction when compared to commands.

Event/Command	Description
`cardEvent`	Contains the raw data of the presented card, acquired from the Wiegand or OSDP interface. The number of valid bits is checked and the timestamp of when the card was presented is recorded.
`idDataPresentedEvent`	`idData` is a representation of an access card using the raw card data, a bit length, and a PIN code. The raw card data is interpreted in accordance with the card formats that have been configured on the network door controller.
`authenticationEvent`	Authentication of the presented card. Checks if the card data is sufficient, unknown, etc. Also checks if more credentials are required (awaiting PIN code) or entered PIN is incorrect.
Send access command	If card is granted access, a command is sent to the relay door which is responsible for unlocking the door.
Portal command	Occurs in the relay door which interprets the received access command. After successful interpretation, a switch command is sent to the relay module.
Send switch command	Occurs in the relay module that toggles the relay, which then unlocks the door.

Table 4.1: Explanation of the card swipe authentication process using events and commands.

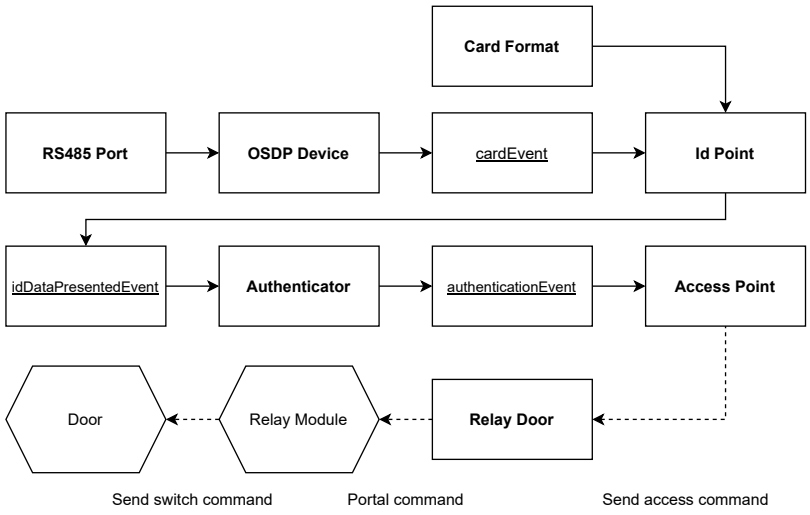


Figure 4.2: Card swipe authentication process overview.

4.3 Measurements

Using the methods described in sections 4.1 and 4.2, the individual changes to the system have been tested. The purpose of testing them individually is to see what parts of our complete solution contribute the most to the power reduction or system latency. As a point of reference, the version of the system that does not include any of our changes is also tested.

Power measurements are done by sampling once every 500 ms in both idle and JMeter use cases. All power usage tests last for a total of two minutes, resulting in 240 samples total. The JMeter test consists of configuring the device with credentials and door relays over a period of approximately one minute, followed by simulated card swipes for the remaining half of the test period. Using the same testing methodology for each test allows us to compare the results between the different solutions more easily.

The latency measurements are done separately from the power measurements. When testing the latency, JMeter is run in five minute long tests, equivalent to approximately 30 simulated card swipes.

The implementation described in section 3.2.4 is used for our CPU Suspend power measurements. Making an isolated test for this implementation is not necessary as the system will be in either a suspended or non-suspended state, where the goal is to spend as much time in a suspended state as possible to maximize potential power savings. Another reason is that this implementation does not directly impact the power usage in the system, as its purpose is to accommodate a system that is now able to enter a suspended state.

Chapter 5
Results

In this chapter, power measurements and system latency results are presented for each of the changes we have implemented, which were described previously in chapter 3. Testing methodology is described in chapter 4. These results are then evaluated in terms of how effective they were in reducing the power consumption and whether they caused any notable increases in system latency.

5.1 Measurements

Here, the measurements are summarized. The power measurement plots recorded during the experiments can be found toward the end of this chapter. Note that the horizontal axis in all plots represents the number of samples, which were taken with a rate of 2 per second. Numbers in parentheses, found in the latency result tables, correspond to the following steps in the authentication process:

```
(1) = cardEvent -> idDataEvent
(2) = idDataEvent -> authenticationEvent
(3) = authenticationEvent -> Access command
(4) = Portal command -> Switch command
```

5.1.1 Original system

What	Min	Max	Average	Percent
(1)	0	0	0	0
(2)	47	66	53	34
(3)	16	41	24	15
(4)	12	59	31	20
Other	33	71	45	31
Total	**135**	**188**	**155**	**100**

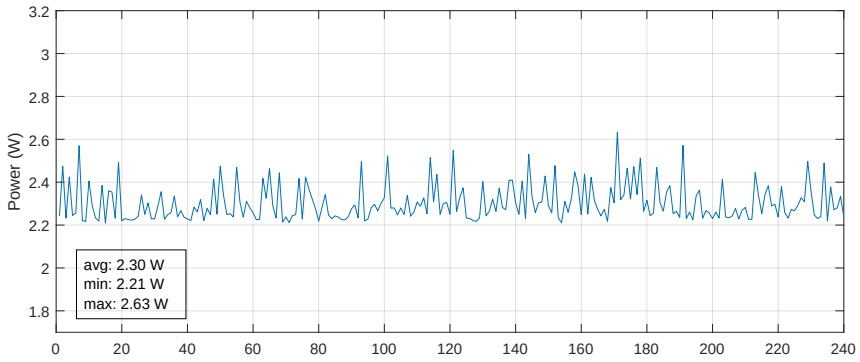
(a) Original system + OSDP (ms).

What	Min	Max	Average	Percent
(1)	0	0	0	0
(2)	59	86	66	37
(3)	19	39	25	14
(4)	13	56	31	17
Other	39	84	53	32
Total	**159**	**204**	**176**	**100**

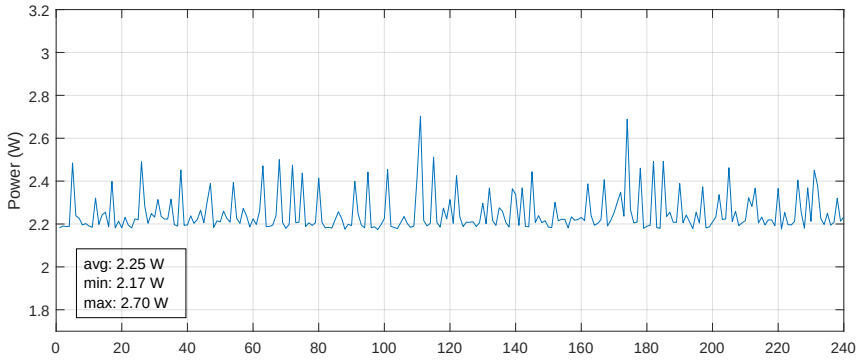
(b) Original system + Wiegand (ms).

Table 5.1: Original system latency results.

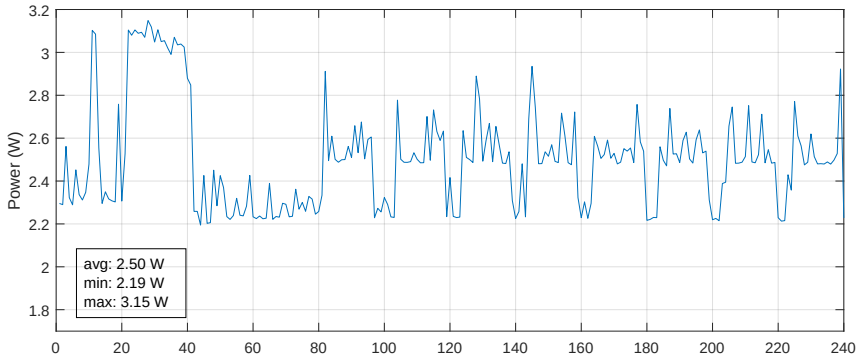
This corresponds to the untouched implementation, i.e. the Axis firmware without any of our modifications applied. Four use cases are presented in Figure 5.1.



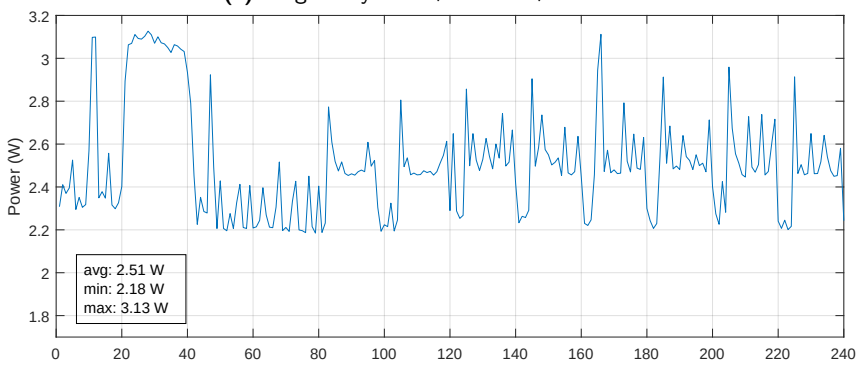
(a) Original system + OSDP + Idle.



(b) Original system + Wiegand + Idle.



(c) Original system + OSDP + JMeter.



(d) Original system + Wiegand + JMeter.

Figure 5.1: Original system power consumption summary.

During idle, the system consumed on average 2.30 W with OSDP and 2.25 W with Wiegand. During JMeter loads the system consumed around 2.5 W for both Wiegand and OSDP. Looking at the standard deviations, idle OSDP was 0.0852, idle Wiegand 0.0919, OSDP JMeter 0.2378, and Wiegand JMeter 0.2457. Looking at the system latency (Table 5.1), the average is 155 ms for OSDP and 176 ms for Wiegand.

5.1.2 `CPUFreq`

What	Min	Max	Average	Percent
(1)	0	1	0	0
(2)	41	61	54	34
(3)	14	40	24	15
(4)	11	48	26	16
Other	39	80	52	35
Total	**141**	**200**	**157**	**100**

(a) `CPUFreq` + OSDP (in milliseconds).

What	Min	Max	Average	Percent
(1)	0	2	0	0
(2)	69	90	75	38
(3)	19	41	26	13
(4)	13	56	33	17
Other	42	91	56	32
Total	**173**	**229**	**193**	**100**

(b) `CPUFreq` + Wiegand (in milliseconds).

Table 5.2: `CPUFreq` latency results.

Enabling `CPUFreq` with the `ondemand` governor, the system drew 2.25 W with OSDP and 2.21 W with Wiegand. Results are shown in Figure 5.2. For the JMeter tests, the average were 2.48 W and 2.45 W for OSDP and Wiegand respectively. Standard deviations for idle tests were 0.1231 and 0.960 for OSDP and Wiegand respectively, whereas the JMeter tests were 0.2562 and 0.2691. As seen in Table 5.2, the latency results were 157 ms and 193 ms for OSDP and Wiegand respectively.

5.1.3 `CPUIdle`

What	Min	Max	Average	Percent
(1)	0	0	0	0
(2)	41	67	54	34
(3)	17	36	24	15
(4)	11	55	33	21
Other	31	59	45	30
Total	**140**	**184**	**157**	**100**

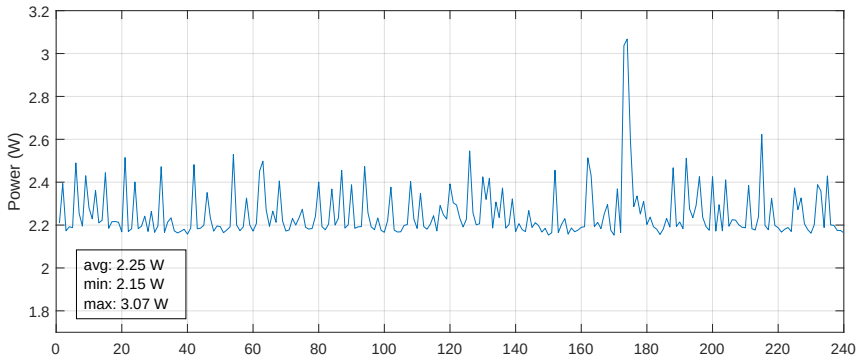
(a) `CPUIdle` + OSDP (in milliseconds).

What	Min	Max	Average	Percent
(1)	0	0	0	0
(2)	26	51	42	29
(3)	18	39	23	15
(4)	11	59	26	18
Other	39	79	51	38
Total	**135**	**184**	**144**	**100**

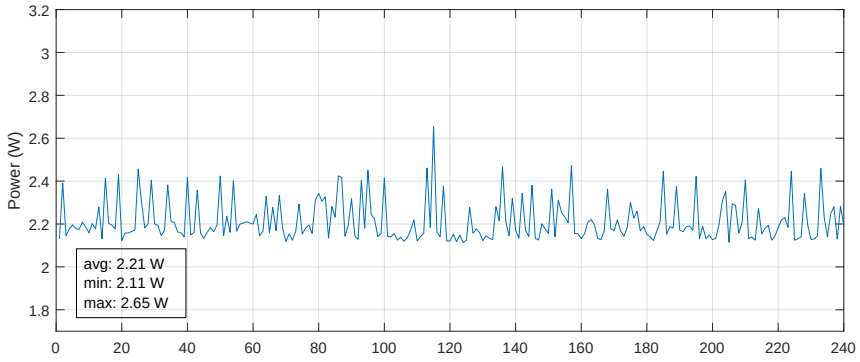
(b) `CPUIdle` + Wiegand (in milliseconds).

Table 5.3: `CPUIdle` latency results.

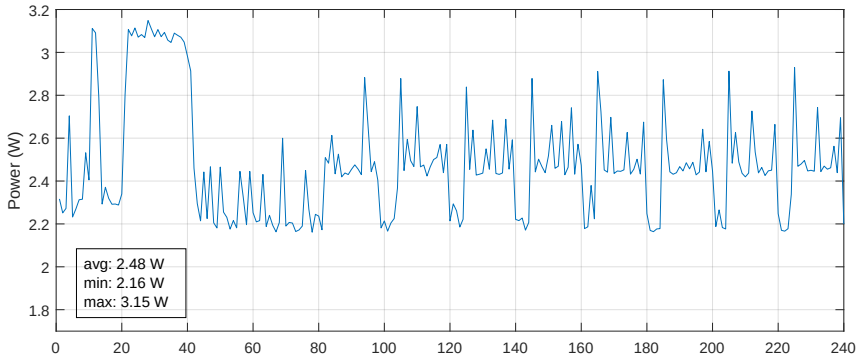
Results for `CPUIdle`, see Figure 5.3. Idle cases for Wiegand and OSDP were near equal at 2.20 W and 2.21 W, respectively. JMeter tests had an equal average power draw of 2.44 W. Standard deviations were 0.081 (OSDP) and 0.1072 (Wiegand) for the idle cases, and 0.2532 and 0.2585 for the respective JMeter tests. In regards to system latency, the average delay was 157 ms for OSDP and 144 ms for Wiegand, see Table 5.3.



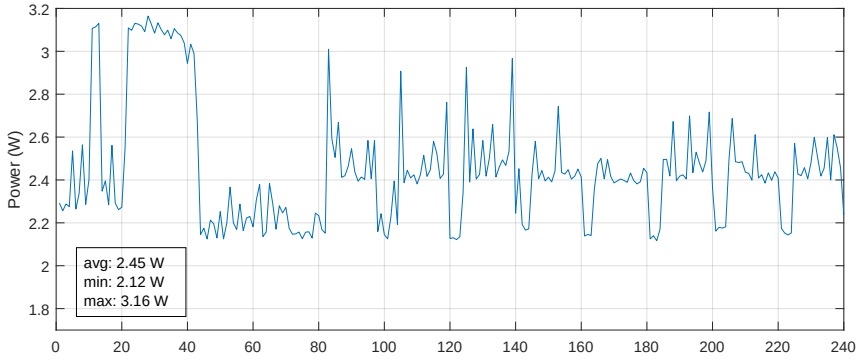
(a) `CPUFreq` + OSDP + Idle.



(b) `CPUFreq` + Wiegand + Idle.

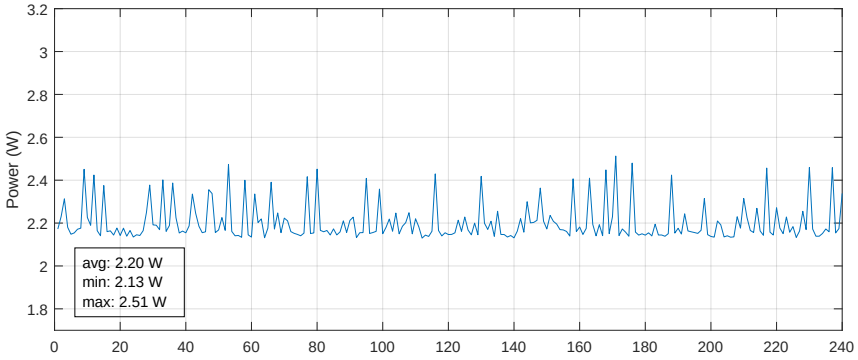


(c) `CPUFreq` + OSDP + JMeter.

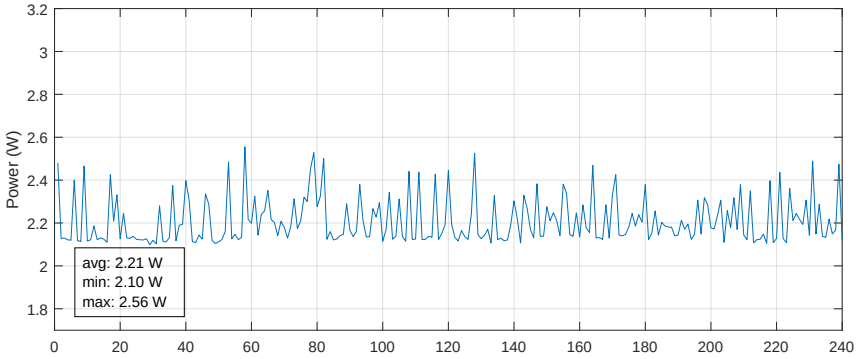


(d) `CPUFreq` + Wiegand + JMeter.

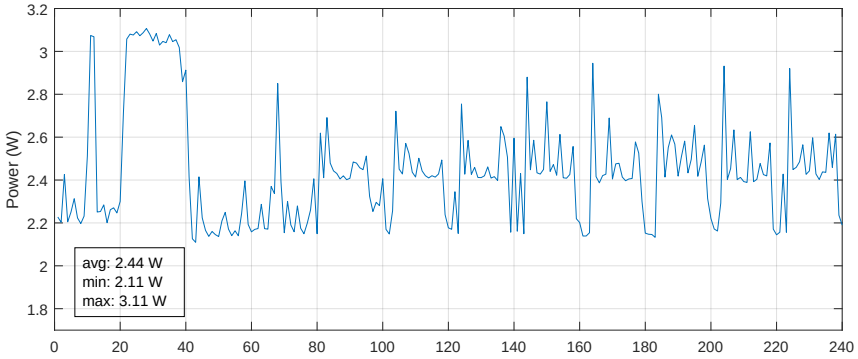
Figure 5.2: `CPUFreq` power consumption summary.



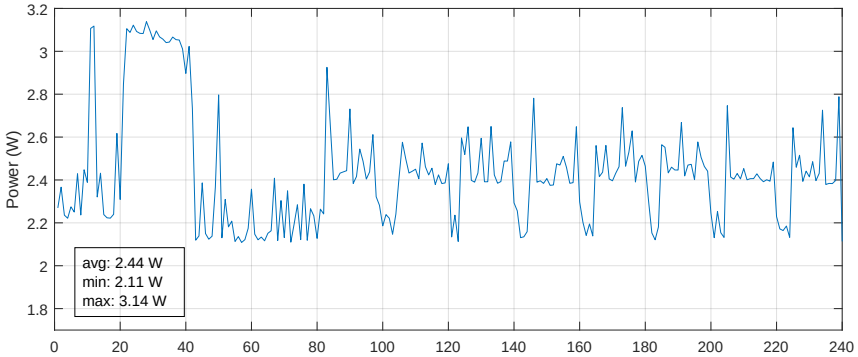
(a) `CPUIdle` + OSDP + Idle.



(b) `CPUIdle` + Wiegand + Idle.

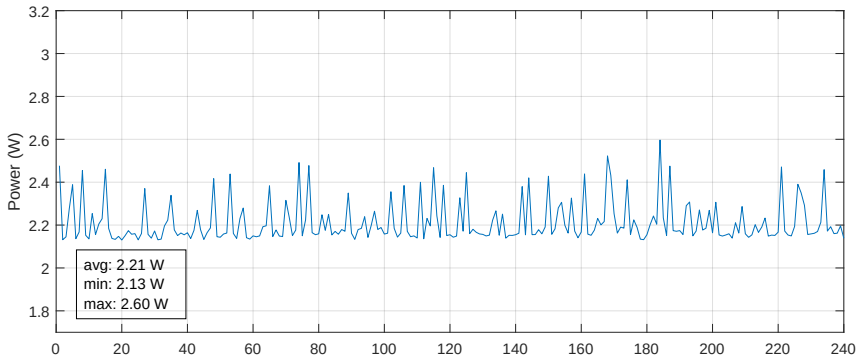


(c) `CPUIdle` + OSDP + JMeter.

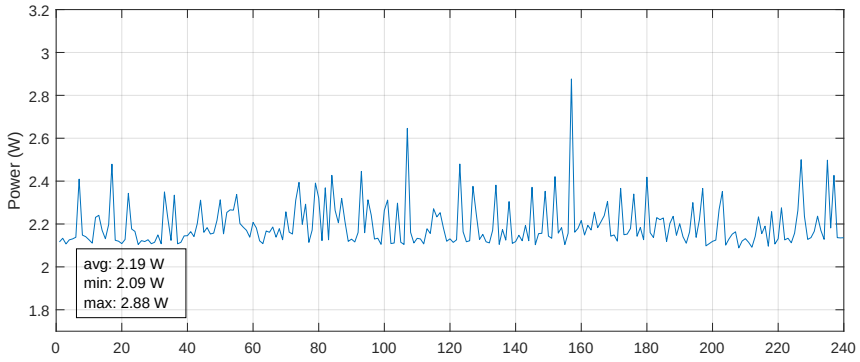


(d) `CPUIdle` + Wiegand + JMeter.

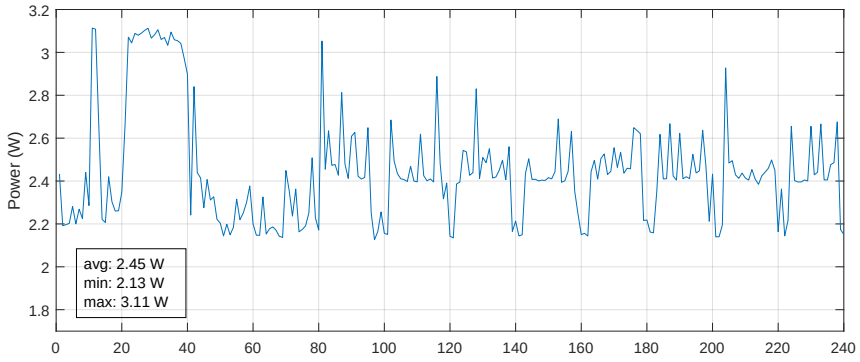
Figure 5.3: `CPUIdle` power consumption summary.



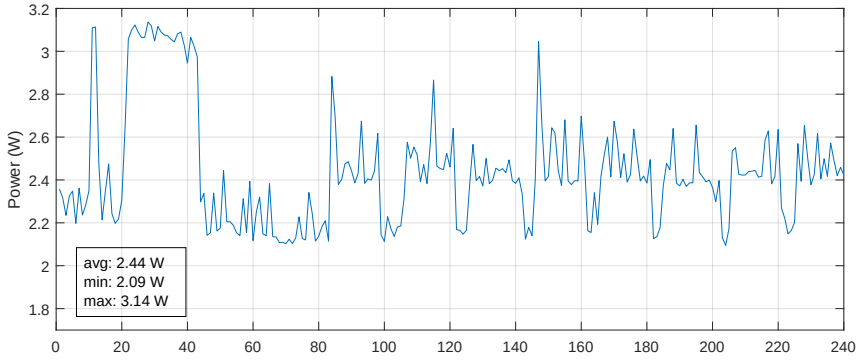
(a) `CPUFreq` + `CPUIdle` + OSDP + Idle.



(b) `CPUFreq` + `CPUIdle` + Wiegand + Idle.



(c) `CPUFreq` + `CPUIdle` + OSDP + JMeter.



(d) `CPUFreq` + `CPUIdle` + Wiegand + JMeter.

Figure 5.4: `CPUFreq` + `CPUIdle` power consumption summary.

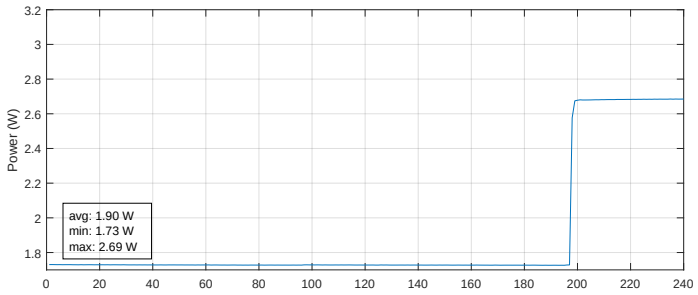


Figure 5.5: CPU suspend idle result.

5.1.4 `CPUFreq` & `CPUIdle`

What	Min	Max	Average	Percent
(1)	0	10	0	0
(2)	43	71	56	34
(3)	17	35	23	14
(4)	12	51	28	17
Other	41	71	53	35
Total	**140**	**193**	**163**	**100**

(a) `CPUFreq` + `CPUIdle` + OSDP (ms).

What	Min	Max	Average	Percent
(1)	0	2	0	0
(2)	39	64	51	32
(3)	20	37	25	15
(4)	11	51	25	15
Other	35	80	54	38
Total	**137**	**182**	**157**	**100**

(b) `CPUFreq` + `CPUIdle` + Wiegand (ms).

Table 5.4: `CPUFreq` + `CPUIdle` latency results.

Having both `CPUFreq` and `CPUIdle` enabled yielded the results found in Figure 5.4. Idle values were 2.21 W and 2.19 W for OSDP and Wiegand respectively. During load the power draws were near equal at 2.45 W for OSDP and 2.44 W for Wiegand. Standard deviations in the idle cases were 0.0852 for OSDP and 0.1045 for Wiegand. The standard deviations for the JMeter tests were 0.2506 for OSDP and 0.2656 for Wiegand. Shown in Table 5.4, the latencies were on average 163 ms for OSDP and 157 ms for Wiegand.

5.1.5 CPU suspend

Lastly, the power measurement results for the system in suspend mode are available in Figure 5.5. Since the system cannot respond to events without exiting suspend mode, we did not test the power consumption in suspend mode under any load. The system was drawing a steady 1.73 W up until about 98 seconds into the test, when the device suddenly started drawing a constant 2.69 W. Between samples 1 and 197, the standard deviation is 0.000841, and between samples 198 and 240, the standard deviation is 0.002146.

The increased power consumption at the end of the test corresponds to a randomly occurring abnormal state where the system does not appear to respond to any input, including the kinds of inputs that normally would cause it to exit suspend mode. It is currently unknown why this occurs.

5.2 Evaluation

Power consumption results will be evaluated first, followed by the system latency.

5.2.1 Power consumption

The power consumption of the network door controller's CPU has been measured by others previously, as presented in section 2.3.6. The main takeaway from these results in this document is that the CPU consumes close to no power when suspended, allowing a significant amount of power to be saved when CPU functionality is not required. As the CPU in the network door controller is always running multiple background processes even when there is no user activity, the document's system idle result is not directly comparable to our idle results. Comparing the results between Figures 5.1a and 5.5, we note a mean difference of $|2.30 - 1.73| = 0.57$ W, implying that the CPU draws approximately 0.57 W ($\approx 25\%$ of the total power usage) in its idle state. This leaves a power consumption of 1.73 W caused by other system components, such as the volatile memory and MCU.

An observation that can be made when comparing the original system with the other measurements (excluding CPU Suspend) is that the power draw varies more when `CPUFreq` or `CPUIdle` is enabled. This is to be expected, as the core idea behind these subsystems is to dynamically switch between higher-power and lower-power CPU modes. The highs in power consumption remain the same, but the lows get lower, which matches the fact that the highest-power modes that can be selected by `CPUFreq` and `CPUIdle` are the modes used in the original system.

Another observation is that a system configured for OSDP on average consumes slightly more power than a system configured for Wiegand. This could be caused by the need to send OSDP poll messages as described in section 2.2.7.

As seen in all figures related to JMeter testing (subfigures c and d), the latter half of the plots have a pattern of energy spikes immediately followed by a valley. This corresponds to the part of the JMeter test where the door is being held open and then closed within a ten second interval. The highs occur when the door is opened and the lows when the door is closed.

5.2.2 Latency

The initial system latencies were 155 ms and 176 ms for OSDP and Wiegand respectively, as seen in Tables 5.1a and 5.1b. Measurements that did not have `CPUIdle` enabled had higher latency in the Wiegand case than in the OSDP case, and otherwise vice versa. For OSDP, the minimum latencies increased in all cases when compared to the test of the original system. Due to variances in the averages, it is difficult to conclude if one implementation has better results than the rest, but at least none of the changes appear to have large impacts on latency.

For this type of system, it is also important to consider the maximum system latencies. If the system has been idle for a while, it will likely be in a lower-powered mode when someone first tries to interact with it, which could potentially lead to a longer latency for this specific user. Looking at Tables 5.1 and 5.4, the disparity between the maximum latencies is small, which reassures us that these Linux subsystems do not negatively impact this particular use case.

Chapter 6
Discussion

This chapter discusses the effectiveness of our implementations, how certain system aspects make power reduction a more challenging task, and whether a battery-powered variant of the system would be feasible.

6.1 Outcome of the implemented improvements

Aside from meeting the requirements of successfully reducing the power consumption and not worsening the latency compared to the original system, it is also important to consider if the system has become more inconvenient to use from an end-user's point of view.

6.1.1 `CPUFreq` & `CPUIdle`

Enabling the `CPUFreq` and `CPUIdle` Linux subsystems led to a combined average reduction in power consumption of about 0.05 W in most test cases, or 0.10 W in the case of an idling system configured to use OSDP. While this is not a huge reduction – only about 2–4% – enabling these two already existing subsystems was relatively simple to do, and even a small reduction in power consumption does help.

We expected the latency after enabling `CPUFreq` and `CPUIdle` to be either unchanged or worse, seeing as enabling them does not do anything to increase the performance of the system.[1] According to our test results, enabling `CPUFreq` made the Wiegand latency worse and enabling `CPUIdle` made the Wiegand latency better, whereas the OSDP latency worsened just a tiny bit with either change. Since we do not have any explanation for why the latency could have improved from enabling `CPUIdle`, we believe that the measured difference in latency mostly is caused by natural variations. There may be a latency impact from enabling these subsystems, but if so, it is a small difference which is unlikely to be a problem for the end-user.

For these reasons combined, we consider enabling `CPUFreq` and `CPUIdle` to be a suitable change to make to the system.

[1] In some systems, in particular thermally constrained ones, decreasing the power consumption can have the side effect of increasing the performance of the system. This network door controller is however not such a system.

6.1.2 CPU suspend

As shown in Figure 5.5, the power consumption when the CPU is suspended is a steady 1.73 W in the normal case. This is an improvement of 21% compared to the system with `CPUFreq` and `CPUIdle` enabled, and an improvement of up to 25% compared to the original system. Compared to the power reduction achieved by enabling `CPUFreq` and `CPUIdle`, this is a much more significant improvement.

The question is whether keeping the system in suspend mode much of the time actually is feasible. Because of the nature of physical access control systems, it is almost certain that an installation of the system will have long periods of no activity, but it is also hard for the system to know in advance when the next request will occur. If the device can easily and quickly wake up from suspend when a request occurs, there is nothing stopping it from being in suspend mode most of the time, but if it is not, the degradation in system functionality may not be worth the savings.

Our implemented solution for waking the system on Wiegand data functions well for waking the device when an end-user interacts with it. From the perspective of a person who wants to unlock a door, the system simply functions as before with no observable difference. While we unfortunately do not have exact numbers for how long the device takes to wake up, as the method we used for measuring the system latency relies on timers which are not running when the CPU is suspended, the time it takes for the system to start running user space code and communicate over Ethernet is so short as to be close to instant to the human eye. Based on this, we believe that the time it takes for the system to wake up is a few tens of milliseconds at most, and possibly much less.

While we were not successfully able to implement OSDP on the MCU, if it were to be implemented, it would likely perform similarly to the Wiegand solution, as the actual method of waking up the system is the same between the two.

Aside from waking up in response to system input, the system must also be able to wake up when accessed over the network. Our solution uses Wake on LAN to accomplish this. Unlike the solutions for Wiegand and OSDP, this is not transparent to the party on the other end – the other party must explicitly send a magic packet before it can start communicating with the door controller. In the case where the other party is an access control server, this requires the server software to be modified. In the case where the other party is a person acting in the capacity of a system administrator, the workflow of that person would need to be modified, which is a more onerous requirement than having to make modifications to a piece of software. With the original system, accessing the door controller is as simple for a system administrator as typing the device's IP address into the address field of a web browser. With a suspended system, the system administrator must take the effort of sending a magic packet, an operation which requires a separate tool from the web browser then used to actually access the door controller's administration interface.

What makes the requirement of sending a magic packet even more cumbersome is the difficulty of sending a magic packet across different networks, as described in section 2.2.4. We anticipate that it is common for an access control server or system administrator to be connected to a different network than the door

controller. Without specifically configuring the networks to allow for sending magic packets or introducing additional devices, it is often not possible to send magic packets to the door controller in such a scenario. This is on its own a big enough problem that suspending the system likely cannot be a feature enabled by default, but rather would have to be an option which the system administrator enables only if sending magic packets to the device is possible with the current network configuration and the system administrator is prepared to do when accessing the door controller's administration interface in the future. We do not anticipate that all system administrators would be willing to go through the effort required.

While the Ethernet controller in the existing system only supports waking up on magic packets, there are other Ethernet controllers which support waking up on other traffic. With such an Ethernet controller, it would be possible to configure the system to wake up on any network traffic intended for it, not just a magic packet. This would require the system to wake up on not only unicast messages directed to it but also broadcast messages, since broadcast messages are used by the Address Resolution Protocol (ARP). This would let a suspended system behave the same as a non-suspended system from the perspective of a system administrator. However, further studies would be needed to determine whether this would cause enough spurious wakeups that suspending the system would not be worth the effort.

6.2 The design of the OSDP protocol

As was described in section 2.2.7, OSDP is a polling protocol. Not only does this result in the non-suspended door controller consuming a little more power when connected to a card reader using OSDP compared to when using Wiegand, as shown in chapter 5, it also requires that the OSDP polling is offloaded from the main CPU if the main CPU is to enter suspend mode. For the system in our work, it is most likely feasible to implement OSDP polling on the already existing MCU, but there may be other physical access control systems which would not have needed an MCU or other low-power secondary processor if it was not for this. Furthermore, if Secure Channel is to be used, the requirements on the MCU would be even higher, as the MCU would need to be able to compute AES-128 encryption and manage associated encryption keys.

Thus, replacing OSDP with a protocol where peripherals can push data to the door controller without the door controller needing to poll would not only result in direct power savings, but would also make it easier to implement suspending the CPU for even greater power savings. However, polling peripherals regularly does have a purpose – it ensures that the connection to the peripherals has not been tampered with. Since the primary purpose of a physical access control system is to prevent break-ins, and since components such as card readers by necessity must be installed in an area accessible to unauthorized persons, this kind of anti-tampering measure is a meaningful feature of an access control system.

In general, we would recommend protocol designers to not rely on polling during the continual operation of a system, due to the impacts it has on power consumption. However, it can be justifiable if it is done for a good reason – in this

case anti-tampering. After all, polling does not make it impossible to implement a system where the CPU enters suspend mode, even though it does make it more complex.

6.3 Encryption in low power systems

When using OSDP Secure Channel, all traffic between reader and controller is encrypted using AES-128, as described previously in section 3.2.5. Since the system MCU is significantly more resource-constrained than the CPU, implementing strong encryption on the MCU for use when the CPU is suspended is a challenging task.

There are two primary classifications of encryption algorithms, referred to as public key encryption and secret key encryption. Public key encryption (also known as asymmetric cryptography) uses a public and a private key, which encrypt and decrypt messages, respectively. Popular public key encryption schemes such as RSA are based on difficult prime number factorization problems and therefore more computationally costly, making public key encryption unsuitable for resource-constrained systems like the network door controller in this thesis. Secret key encryption schemes use only a single key for both encryption and decryption. The AES-128 standard, mentioned previously, is a commonly found cryptographic system which is classified as a secret key encryption scheme. If implemented in hardware, the implementation is up to 10-100 times faster than the corresponding software implementation [58, 59].

In terms of flexibility and maintainability, software implementations are better as the software (firmware) can be updated during the development process simply by flashing the updated firmware, whereas hardware implementations would require a physical reconstruction of the circuit – a time-consuming and costly endeavor that could potentially increase the size of the circuit board itself. Because a chosen encryption method typically does not get changed often, the lack of flexibility is not a concern for this type of application.

To account for the MCU's limited computation capabilities, it is optimal to use a hardware-based implementation for encryption and other common functions. The microcontroller used in the network door controller today does not have support for hardware encryption. It is advantageous to use an MCU with hardware support for e.g. AES-128 encryption to conserve both power consumption and system latency. As previously mentioned in section 3.2.5, a software implementation is feasible if the amount of data that is to be encrypted is small. The card credentials sent from the card reader to the door controller are as small as a few bytes, and the total size of an OSDP message including headers and footers is also fairly small (less than a hundred bytes). However, as will be described in greater detail in section 6.4, building a battery-based power solution requires all areas of the system to consume minimal power, which hardware implementations are well suited for.

6.4 Creating a battery-powered system

If a product variant which does not need to be connected to the power grid is to be created, the system must have a battery of sufficient capacity in relation to the system's power draw. Otherwise, the battery will need to be replaced frequently, wasting natural resources and creating a maintenance burden that likely is worse than the burden of connecting the system to the power grid during the initial installation.

To calculate the best-case battery life $T_{battery}$ (in hours) for our system using a given battery [60], we use the following equation:

$$T_{battery} = \frac{C_{battery} \cdot V_{battery}}{W_{device}} \tag{6.1}$$

where $C_{battery}$ is the battery's capacity expressed in ampere hours (Ah), $V_{battery}$ the nominal voltage (V) of the battery, and W_{device} is the system's power consumption in watts (W). In the best case, i.e. during CPU suspension, the power consumption was 1.73 W (see Figure 5.5), thus $W_{device} = 1.73$ W.

Battery	**Anode (−)**	**Cathode (+)**	**Nominal Voltage (V)**	**Energy density (MJ/kg)**	**Special characteristics**
Alkaline	Zinc	Manganese dioxide	1.5	0.5	Long shelf life, supports high-to-medium-drain applications
Zinc-carbon	Zinc	Manganese dioxide	1.5	0.13	Economical in cost per hour for low current consumption
Lithium (BR)	Lithium	Carbon monofluoride	3	1.3	Wide temperature operation, high internal impedance, low pulse current
Lithium (CR)	Lithium	Manganese dioxide	3	1	Good pulse capability, stable voltage during discharge
Lithium-thionyl chloride	Lithium	Sulfur-oxygen chlorine	3.6	1.04	Low self-discharge rate, can support 20-year battery
Zinc-air	Zinc	Oxygen	1.4	1.69	High energy density, battery life of weeks to months

Table 6.1: Common battery types in mainstream embedded-system applications.

Characteristics of battery types commonly found in embedded systems are listed in Table 6.1 [61]. One thing to note about these batteries are their low nominal voltages. During our testing (as previously described in section 4.1), we used a power supply of 48 V for powering the network door controller, which eliminates the need of using a step-up converter: a device that raises the incoming voltage to levels that different components of the system require to operate. The main issue with using a step-up converter is that this process is more inefficient compared to the analogous step-down converter, as the converter itself requires power to function, resulting in higher power consumption [62]. To work around this, battery cells can be stacked in series to achieve higher voltages and capacity.

Unfortunately, the power consumption of the system in our study is too high for battery operation to be reasonable even after our improvements. Assume that we install four lithium-thionyl cloride batteries (see Table 6.1) in series, giving us a nominal voltage of $4 \cdot 3.6 = 14.4$ V. Then there would be no need for a step-up converter as most system components operate at or below 12 V. Also assume that every battery has a capacity of 8.5 Ah, giving us a total capacity of $4 \cdot 8.5 = 34$ Ah. Inserting these values into equation 6.1, we get a battery life of 283 hours (less than 12 days) if the power draw is a constant 1.73 W. This would mean that the batteries would have to be replaced over 30 times a year, which would be a very tedious (and expensive) task for system administrators.

To build a power efficient system, all components must be built and implemented with power consumption in mind. Looking at modern smartphones, such as the iPhone 11 with its 3.11 Ah lithium-ion battery, it can play video for up to 17 hours, or music for 65 hours, before the battery becomes discharged [63]. This is possible on a battery that is 11 times smaller than our hypothetical 12 day battery. If one were to replace the iPhone battery with a 34 Ah battery, it would be the equivalent of 186 hours of video playback, or 710 hours of playing music. Keep in mind that these comparisons are made between a smartphone under load, versus a system that is idling, showing that an iPhone constantly playing music outperforms an idle network door controller with a suspended CPU in terms of power consumption. Because of this and the already short battery life discussed previously, we believe that there is still a lot that has to be done in terms of power optimizations before a battery-powered variant of the network door controller could be implemented and realized.

6.5 Replacing Ethernet with Wi-Fi

The main reason for wanting to make the system battery-powered is to remove the need to run cables for it, reducing installation costs. If making a battery-powered system becomes feasible in the future, one major obstacle remains regarding cables: the system currently relies on being able to connect to a network using Ethernet. To make the system as simple to install as possible, communication with the network would need to be solved by other means, e.g. by installing Wi-Fi technology on the network door controller. However, how much more power does Wi-Fi require, and how can it be minimized?

To operate on a wireless LAN, a radio network interface controller (NIC) must be installed on the device, which provides wireless connectivity via 802.11a or 802.11b/g [64], which are IEEE standards for transmitting data over a wireless network [65]. According to a benchmark [66] performed on the Raspberry Pi 3 Model B, a low-power single-board computer [67], Ethernet and Wi-Fi require 2 mA and 20 mA of power respectively, implying that a wireless connection requires up to ten times as much power to operate than a wired connection. It is worth noting that this Raspberry Pi system consumes 300 mA during idle, meaning that the radio NIC is responsible for approximately 7% of the total system power consumption at most. If one is to consider not only the extra power consumption on the network door controller but also the necessary equipment to facilitate a

wireless solution, such as access points and routers, then the energy efficiency contribution from a sustainability and climate point of view is further reduced.

This increase in power can potentially be mitigated by using a lightweight communication protocol. Message Queuing Telemetry Transport (MQTT) is such a protocol, which is well-suited for communications between low-power devices. This protocol uses a publish/subscribe scheme, i.e. it is a protocol where the *publisher* (e.g. a server) distributes information that *subscribers* (the network door controllers) choose to acquire [68]. For example, the network door controller could choose to subscribe to the central server to receive updates to its local credentials database. This scheme may not be perfectly suited for network door controllers, as the credentials published by the central server may not be applicable to all door controllers in an installation. Assuming that the resource costs for multiple MQTT publish/subscribe instances is not too costly, every controller could be part of their own unique MQTT instance with the central server, to receive updated credentials specific to that controller, in a power efficient and wireless way.

Chapter 7
Future Work

This chapter contains suggestions for future projects which can further reduce the power consumption of the network door controller.

7.1 Hardware

This thesis has focused on power consumption optimizations achievable through software changes, but there are also multiple areas of hardware changes which could be explored for the purpose of reducing the system's power consumption.

7.1.1 RAM power consumption

One component which we could not fully power down during idle periods is the random access memory. A significant portion of the power used by a typical mobile device can be attributed to the memory system, and different types of memory have varying performance and power characteristics [69]. Perhaps switching the type of memory used in the device could lead to the system consuming less power.

7.1.2 Ambient energy sources

Another avenue of exploration is whether the amount of power drained from the power grid or battery can be reduced not by reducing the power consumption of the device but by supplementing the primary power source with an ambient source of energy ("energy harvesting"). Such sources include solar energy, thermal energy, and radio frequency energy [70, 71]. Unfortunately, most of these sources provide rather low amounts of power, and the one which provides the most power, solar energy, is difficult to make use of since door controllers are normally installed indoors. We do not anticipate these sources of energy to be able to provide the majority of the system's power even if the system is made more energy efficient in the future, but for a very efficient system, ambient sources may be able to contribute a meaningful amount of power.

7.1.3 Waking up on more types of Ethernet activity

As previously mentioned in section 6.1.2, using an Ethernet controller which allows the system to wake up on more types of network activity than just magic packets

could make it possible to make system suspend transparent to system administrators. Studies are needed to determine whether this would cause the system to wake up too frequently for suspending to be worth it.

7.2 Software

In addition to the software changes made during this thesis, work remains to make the system's software accommodate suspending. Other suggestions for software changes which can reduce the power consumption are also made in this section.

7.2.1 Automatically suspend or wake up CPU

Since suspending the CPU saves significant amounts of power, implementing ways for the system to automatically suspend the CPU is important. One possible implementation is a timer which suspends the system once a certain time has passed since e.g. the last card swipe or the last received Ethernet packet. An alternative, more sophisticated implementation would be to suspend the CPU using external system events. For example, if there are surveillance cameras installed nearby, they could detect whether any person can be seen and send a signal to suspend the door controller when nobody is present. Of course, this only saves power if the cameras would be installed regardless of this power saving feature, since the cameras consume power in themselves. For Axis, using surveillance cameras in this manner could act as a special feature that is available between Axis network cameras and network door controllers, as they have the opportunity to make the necessary implementations on both devices.

Nearby cameras could also be used to preemptively wake up the system when a person is seen approaching the door. This would ensure that any latency associated with waking up is not noticed by the user. While this latency is not very large in our implementation, there is the possibility that future implementations could be able to enter deeper sleep states that save even more power at the expense of additional latency, in which case preemptively waking up the system would noticeably improve the user experience.

7.2.2 Wake up CPU for reasons other than interaction

On the network door controller, there are actions that run periodically without external involvement. An example of such an action is to check to state of I/O pins. Allowing the device to wake up by itself is important for preserving this type of device functionality. Policies for determining when to wake up could be implemented on the MCU, as it is never in a suspended state and can wake up the CPU by sending data to it over UART. As an example, the CPU could be made to wake up for a short moment periodically throughout the day. Another alternative is to queue up these types of events for when the CPU wakes up by other means, i.e. from Ethernet or card reader activity. However, this risks slowing down the authentication process for the user if a large number of events are in the queue, and could lead to events being processed significantly later than is ideal.

7.2.3 Replace HTTP with CoAP

As described previously in section 1.3.2, CoAP is a simple web transfer protocol that is suited for low-power networks containing constrained devices, which integrates easily with HTTP. Additionally, an Apache web server (see section 2.3.4) runs on the network door controller to allow administrative access over HTTP. This thesis did not investigate the power consumption of when a user is interacting with the Apache web server. Isolated power measurement tests between HTTP and CoAP could help determine if the protocol's power savings are significant enough to consider changing protocols in a future version of the system.

Chapter 8
Conclusions

The primary objective of this thesis was to reduce the power consumption of an Axis network door controller without compromising system latencies in a significant way. In our research and implementation, several suggestions have been proposed and evaluated, with varying degrees of complexity.

The simplest of the implemented improvements is to enable the `CPUFreq` and `CPUIdle` Linux subsystems on the network door controller's CPU. Experiment results showed that the average power consumption during idle workloads can be reduced by approximately four percent. While not huge, it is a meaningful improvement for this type of system which spends a lot of time idling.

The idle power consumption can be reduced much further, by as much as 25% compared to the original system, by maximizing the amount of time the CPU is suspended. Doing this without compromising the primary functions of the system requires delegating certain critical system functionality to the MCU and implementing policies for suspending and waking up the CPU when appropriate. We have implemented many of the required modifications in our proof of concept, but additional work remains to be done, in particular supporting OSDP polling while suspended and implementing policies for when to suspend. If all proposed modifications are successfully implemented, the system will be able to suspend and wake up automatically without negatively affecting end-users, but system administrators may need to change their workflow.

In the end, the improvements in this thesis did not reduce the power consumption enough to make a battery-powered system feasible. There are however various avenues of investigation for future power savings, both ones involving hardware modifications as well as software-only improvements like using more efficient communication protocols.

References

[1] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan, "Powering the internet of things," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*. Association for Computing Machinery, 2014, pp. 375—380.

[2] M. Alam, K. A. Shakil, and S. Khan, *Internet of Things (IoT): Concepts and Applications*, 1st ed. Cham, Switzerland: Springer International Publishing, 2020.

[3] F. Sant'Anna, A. Sztajnberg, A. L. de Moura, and N. Rodrigues, "Transparent standby for low-power, resource-constrained embedded systems: a programming language-based approach (short WIP paper)." *ACM SIGPLAN Notices - LCTES '18*, vol. 53, no. 6, pp. 94 – 98, 2018.

[4] C. B. Z. Shelby, K. Hartke and B. Frank, "RFC 7252: The constrained application protocol (CoAP)," Internet Engineering Task Force, Tech. Rep., 2014. [Online]. Available: https://tools.ietf.org/html/rfc7252 (Accessed 2021-04-20).

[5] S. Hoppe and A. Stark. (2019) IoT basics: What is OPC UA? Spotlight Metal. [Online]. Available: https://www.spotlightmetal.com/iot-basics-what-is-opc-ua-a-842878/ (Accessed 2021-04-21).

[6] Y. Wang, C. Pu, P. Wang, and J. Wu, "A CoAP-based OPC UA transmission scheme for resource-constrained devices," in *2020 Chinese Automation Congress (CAC)*, 2020, pp. 6089–6093.

[7] W. Y. Ng, W. Pedrycz, and D. S. Yeung. Machine learning. IEEE SMC. [Online]. Available: https://www.ieeesmc.org/technical-activities/cybernetics/machine-learning (Accessed 2021-04-28).

[8] M. Kaur. (2019, May) Top 10 real-life examples of machine learning. Big Data Made Simple. [Online]. Available: https://bigdata-madesimple.com/top-10-real-life-examples-of-machine-learning (Accessed 2021-04-28).

[9] C. Leech, Y. P. Raykov, E. Ozer, and G. V. Merrett, "Real-time room occupancy estimation with bayesian machine learning using a single pir sensor and microcontroller," in *2017 IEEE Sensors Applications Symposium (SAS)*, 2017, pp. 1–6.

[10] D. Duarte, N. Vijaykrishnan, M. J. Irwin, H.-S. Kim, and G. McFarland, "Impact of scaling on the effectiveness of dynamic power reduction schemes," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 2002, pp. 382–387.

[11] K. De Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho, "The energy/frequency convexity rule: Modeling and experimental validation on mobile devices," in *Parallel Processing and Applied Mathematics*. Springer Berlin Heidelberg, 2014, pp. 793–803.

[12] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, 1992.

[13] V. Venkatachalam and M. Franz, "Power reduction techniques for microprocessor systems," *ACM Computing Surveys*, vol. 37, no. 3, pp. 195–237, Sep. 2005.

[14] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 international conference on Power aware computing and systems*, 2010, pp. 1–8.

[15] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Mobile Computing*. Springer, 1994, pp. 449–471.

[16] S. Dawson-Haggerty, A. Krioukov, and D. E. Culler, "Power optimization – a reality check," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-140, Oct 2009. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-140.html (Accessed 2021-02-22).

[17] Y.-P. You, C. Lee, and J. K. Lee, "Compilers for leakage power reduction," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, no. 1, pp. 147–164, 2006.

[18] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini, "Application transformations for energy and performance-aware device management," in *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2002, pp. 121–130.

[19] A. Sinha and A. Chandrakasan, "Dynamic power management in wireless sensor networks," *IEEE Design & Test of Computers*, vol. 18, no. 2, pp. 62–74, 2001.

[20] B. Mangione-Smith, "Low power communications protocols: paging and beyond," in *1995 IEEE Symposium on Low Power Electronics. Digest of Technical Papers*. IEEE, 1995, pp. 8–11.

[21] Optimize for Doze and App Standby. Android Developers. [Online]. Available: https://developer.android.com/training/monitoring-device-state/doze-standby.html (Accessed 2021-04-22).

[22] P. Turner. (2016, Jul) Diving into Doze Mode for developers. Big Nerd Ranch. [Online]. Available: https://www.bignerdranch.com/blog/diving-into-doze-mode-for-developers/ (Accessed 2021-04-22).

[23] R. J. Wysocki. (2017) CPU performance scaling. The Linux Kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/v4.19/admin-guide/pm/cpufreq.html (Accessed 2021-01-28).

[24] D. Brodowski, N. Golde, R. J. Wysocki, and V. Kumar. CPUFreq governors. The Linux Kernel documentation. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt (Accessed 2021-02-17).

[25] R. J. Wysocki. (2018) CPU idle time management. The Linux Kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/v5.0/admin-guide/pm/cpuidle.html (Accessed 2021-01-29).

[26] J. Corbet. The cpuidle subsystem. LWN.net. [Online]. Available: https://lwn.net/Articles/384146/ (Accessed 2021-03-31).

[27] System sleep states – the Linux kernel documentation. The Linux Kernel. [Online]. Available: https://www.kernel.org/doc/html/v4.19/admin-guide/pm/sleep-states.html (Accessed 2021-03-30).

[28] Power management/suspend and hibernate - ArchWiki. Arch Linux. [Online]. Available: https://wiki.archlinux.org/index.php/Power_management/Suspend_and_hibernate (Accessed 2021-03-31).

[29] P. Luberus and A. Nyandoro, "Implementing wake-on-LAN in institutional networks," *Journal of Applied Business and Economics*, vol. 16, no. 1, pp. 66–73, 2014.

[30] (1995) Magic packet technology. AMD. [Online]. Available: https://www.amd.com/system/files/TechDocs/20213.pdf (Accessed 2021-03-04).

[31] M. Popa and T. Slavici, "Embedded server with Wake on LAN function," in *IEEE EUROCON 2009*. IEEE, 2009, pp. 365–370.

[32] J. A. Gil-Martínez-Abarca, F. Maciá-Pérez, D. Marcos-Jorquera, and V. Gilart-Iglesias, "Wake on LAN over internet as web service," in *2006 IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2006, pp. 1261–1268.

[33] G. Stoitsov and Z. Guglev, "Server for remote generation of Wake on LAN packets in the LAN," *International Journal of Recent Development in Engineering and Technology*, vol. 3, no. 2, pp. 6–27, 2014.

[34] Wake-on-LAN. Arch Linux. [Online]. Available: https://wiki.archlinux.org/index.php/Wake-on-LAN (Accessed 2021-03-05).

[35] A. von Nagy. (2010) Wake on Wireless LAN. Revolution Wi-Fi. [Online]. Available: http://revolutionwifi.blogspot.com/2010/11/wake-on-wireless-lan.html (Accessed 2021-03-05).

[36] N. Mishra, K. Chebrolu, B. Raman, and A. Pathak, "Wake-on-WLAN," in *Proceedings of the 15th International Conference on World Wide Web*. Association for Computing Machinery, 2006, pp. 761–769.

[37] S. Campbell. (2016) Basics of UART communication. Circuit Basics. [Online]. Available: https://www.circuitbasics.com/basics-uart-communication/ (Accessed 2021-04-19).

[38] Y.-Y. Fang and X.-J. Chen, "Design and simulation of UART serial communication module based on VHDL," in *3rd International Workshop on Intelligent Systems and Applications*. IEEE, 2011, pp. 1–4.

[39] T. Kugelstadt. (2008, Feb) The RS-485 design guide. Texas Instruments. [Online]. Available: https://www.ti.com/lit/an/slla272c/slla272c.pdf (Accessed 2021-04-13).

[40] *TIA-485, Revision A, March 1998 - Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems*, Telecommunications Industry Association, Dec 2012.

[41] U. Brckelmann and T. Salazar. (2000, Oct) Trim the fat off RS-485 designs. EE Times. [Online]. Available: https://www.eetimes.com/trim-the-fat-off-rs-485-designs (Accessed 2021-04-24).

[42] Quick reference for RS485, RS422, RS232 and RS423. RE Smith. [Online]. Available: http://www.rs485.com/rs485spec.html (Accessed 2021-05-21).

[43] Open Supervised Device Protocol (OSDP). Security Industry Association. [Online]. Available: https://www.securityindustry.org/industry-standards/open-supervised-device-protocol (Accessed 2021-04-12).

[44] *Electronic access control systems – Open supervised device protocol (OSDP)*, International Electrotechnical Commission, 2019, IEC 60839-11-5.

[45] (2006) Understanding card data formats. HID Corporation. [Online]. Available: https://www.hidglobal.com/sites/default/files/hid-understanding_card_data_formats-wp-en.pdf (Accessed 2021-04-15).

[46] *Access Control Standard | Protocol for the 26-bit Wiegand Reader Interface*, Security Industry Association, 1996.

[47] systemd. systemd.io. [Online]. Available: https://systemd.io/ (Accessed 2021-04-30).

[48] D. Miessler. (2019, Dec) The difference between system v and systemd. [Online]. Available: https://danielmiessler.com/study/the-difference-between-system-v-and-systemd (Accessed 2021-04-30).

[49] systemd/timers. Arch Linux. [Online]. Available: https://wiki.archlinux.org/index.php/Systemd/Timers (Accessed 2021-04-30).

[50] FreeRTOS. [Online]. Available: https://www.freertos.org/ (Accessed 2021-02-08).

[51] R. Bannatyne and G. Viot, "Introduction to microcontrollers," in *WESCON/97 Conference Proceedings*. IEEE, 1997, pp. 564–574.

[52] Support for the ARM power state coordination interface (PSCI). kernelconfig.io. [Online]. Available: https://www.kernelconfig.io/config_arm_psci (Accessed 2021-03-29).

[53] "`phy_suspend`," The Linux kernel, version 4.19. [Online]. Available: https://github.com/torvalds/linux/blob/v4.19/drivers/net/phy/phy_device.c#L1180-L1183 (Accessed 2021-03-29).

[54] "`mdio_bus_phy_may_suspend`," The Linux kernel, version 4.19. [Online]. Available: https://github.com/torvalds/linux/blob/v4.19/drivers/net/phy/phy_device.c#L96-L97 (Accessed 2021-03-29).

[55] G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin, "Efficient software implementation of AES on 32-bit platforms," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2002, pp. 159–171.

[56] J. Daemen and V. Rijmen, "Efficient block ciphers for smartcards," in *USENIX Workshop on Smartcard Technology*, 1999.

[57] Apache JMeter. The Apache Software Foundation. [Online]. Available: https://jmeter.apache.org/ (Accessed 2021-05-11).

[58] B. Furht, Ed., *Encyclopedia of Multimedia*. Berlin, Heidelberg: Springer-Verlag, 2006.

[59] P. Kitsos, O. Koufopavlou, G. Selimis, and N. Sklavos, "Low power cryptography," in *Journal of Physics: Conference Series*, vol. 10. IOP Publishing, Jan 2005, pp. 343–347.

[60] Power capacity and power capability. Adafruit. [Online]. Available: https://learn.adafruit.com/all-about-batteries/power-capacity-and-power-capability (Accessed 2021-06-01).

[61] Selecting the right battery for your embedded application. Embedded.com. [Online]. Available: https://www.embedded.com/selecting-the-right-battery-for-your-embedded-application/ (Accessed 2021-06-01).

[62] G. A. Rincón-Mora and N. Keskar. (2006, Aug) Unscrambling the power losses in switching boost converters. EE Times. [Online]. Available: https://www.eetimes.com/unscrambling-the-power-losses-in-switching-boost-converters/ (Accessed 2021-06-01).

[63] Apple iPhone 11 - full phone specifications. GSMArena.com. [Online]. Available: https://www.gsmarena.com/apple_iphone_11-9848.php (Accessed 2021-06-01).

[64] J. Geier, *Wireless Networks First-Step*, 1st ed. Indianapolis, Indiana, USA: Cisco Press, 2004.

[65] P. Christensson. 802.11a. TechTerms. [Online]. Available: https://techterms.com/definition/80211a (Accessed 2021-06-07).

[66] J. Bainbridge. Raspberry Pi power. [Online]. Available: https://github.com/superjamie/lazyweb/wiki/Raspberry-Pi-Power (Accessed 2021-06-07).

[67] Raspberry Pi 3 Model B. Raspberry Pi. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b (Accessed 2021-06-07).

[68] MQTT – the standard for IoT messaging. [Online]. Available: https://mqtt.org/ (Accessed 2021-05-24).

[69] R. Duan, M. Bi, and C. Gniady, "Exploring memory energy optimizations in smartphones," in *2011 International Green Computing Conference and Workshops*, 2011, pp. 1–8.

[70] S. Kim, R. Vyas, J. Bito, K. Niotaki, A. Collado, A. Georgiadis, and M. M. Tentzeris, "Ambient RF energy-harvesting technologies for self-sustainable standalone wireless sensor platforms," *Proceedings of the IEEE*, vol. 102, no. 11, pp. 1649–1666, 2014.

[71] B. Allen, T. Ajmal, V. Dyo, and D. Jazani, "Harvesting energy from ambient radio signals: a load of hot air?" in *2012 Loughborough Antennas & Propagation Conference (LAPC)*. IEEE, 2012, pp. 1–4.

Printed by Tryckeriet i E-huset, Lund 2021

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2021-826
http://www.eit.lth.se