Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Design and implementation of embedded components of an RFID access control system

Bachelor's Thesis

2018
Adam Dej

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Design and implementation of embedded components of an RFID access control system

Bachelor's Thesis

Študijný program:	Informatics
Študijný odbor:	2508 Informatics
Školiace pracovisko:	Department of Computer Science
Školiteľ:	RNDr. Richard Ostertág, PhD.

Bratislava, 2018
Adam Dej

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Adam Dej
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Design and implementation of embedded components of an RFID access control system
Návrh a realizácia vstavaných komponentov RFID systému pre kontrolu prístupu

Anotácia: Hlavné ciele tejto práce sú:
* Dizajn, implementácia, testovanie a integrácia vložených komponentov systému
* Dizajn komunikačných protokolov pre použitie s vloženými komponentami systému
* Zlepšenie existujúcich častí systému na strane servera
* Integrácia existujúcich komponentov servera s vloženými komponentami systému
* Automatizované testovanie celého systému
* Nasadenie systému

Vedúci: RNDr. Richard Ostertág, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 26.10.2016

Dátum schválenia: 26.10.2016 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgements:

I would like to thank RNDr. Richard Ostertág, PhD. and Mgr. Tomáš Vinař, PhD. for advice regarding the system design and for tools and material support required for development of these components.

Next, I would like to thank Ľubomír Kučera for insightful discussions that brought an outside perspective to the project.

I would also like to thank Mário Hodas for feedback and advice about hardware and firmware design.

Finally, I would like to thank Štefan Kolek (and company Devin Solutions) for covering development costs of this project.

Abstract

Embedded components of project Deadlock are devices responsible for authentication of users (using e.g. RFID cards, PINs, …) and control of number of Points of Access (e.g. doors, printers, …). There are two main types of embedded components: Reader and Controller. These devices communicate with the Deadlock Server, forming a complete Deadlock System. They are designed to be reliable, modular, maintainable and easy to produce. They are well documented, fully open-source and open-hardware.

This thesis will first describe the architecture of the Project Deadlock. Then hardware design of Reader and Controller is described. It then focuses on the implementation of firmware for the Reader component.

Next, communication interfaces and protocols used by Reader and Controller are described, followed by the description of implementation of Controller firmware.

Finally, we explain software testing methodology used during development. We conclude with the summary of achieved results.

Keywords: access control, embedded devices design, network protocol design

Abstrakt

Vnorené komponenty projektu Deadlock sú zariadenia zodpovedné za autentifikáciu používateľa (napríklad pomocou RFID kariet, PIN kódov, …) a za umožnenie vstupu do miestností alebo ovládania ostatných zariadení. Existujú dva typy takýchto zariadení: Reader (čítačka) a Controller (ovládač). Tieto zariadenia, spolu s Deadlock serverom tvoria kompletný systém Deadlock. Tieto zariadenia sú spoľahlivé, modulárne, jednoduché na údržbu a jednoduché na výrobu. Sú dobre zdokumentované a kompletne open-source a open-hardware.

Táto práca začína opisom architektúry projektu Deadlock. Ďalej opisuje hardvérový dizajn Readera a Controllera. Následne opisuje implementáciu firmvéru komponentu Reader.

Potom sa venuje opisu komunikačných rozhraní a protokolov medzi Readerom a Controllerom. Nasleduje opis implementácie firmvéru Controllera.

Záverom práca ukazuje metodológiu testovania softvéru použitú počas vývoja. Práca je zakončená popisom dosiahnutých výsledkov.

Kľúčové slová: kontrola prístupu, dizajn vnorených zariadení, dizajn sieťových protokolov

Table of Contents

Introduction

Several institutions, such as schools or universities, need to be able to allow a group of people in a specified time window to access protected rooms (such as computer laboratories). This requirement is best fulfilled by an electronic Physical Access Control System, capable of unlocking doors and logging access attempts. Many different systems that implement this functionality are available commercially, however their cost per door is usually prohibitive, and are difficult to integrate with existing systems, limiting their usability. Free systems, on the other hand, lack sufficient quality and reliability.

Therefore Project Deadlock was created to bridge this gap. The goal of Project Deadlock is to create an open-source and open-hardware physical access control system. It is designed to be reliable, extensible, and cost-effective both to manufacture and to maintain.

This thesis provides an introduction to Project Deadlock and its overview in Chapter 1. This chapter describes requirements and design principles developed jointly by Student Development Team, and references previously published works on this project. It introduces embedded component types "Reader" and "Controller", and explains responsibilities of each component.

Rest of the thesis focuses on author's contribution to the project. We describe hardware design of embedded devices used in this project in chapter 2. Chapter 3 focuses on description of embedded firmware for the Reader component, which is responsible for authenticating the user. Next, in chapter 4 we describe how embedded components communicate with each other on multiple layers, and the protocol stack that is used for this purpose. Then, in chapter 5, design of firmware for the Controller component is described. We conclude the thesis with description of methods used for testing various software components.

Chapter 1

System Overview

The goal of Project Deadlock is to create a complete physical access control system. This system was initially designed to primarily open doors, but several different types of physical Points of Access (PoAs) may be guarded by this system. A user wishing to get access to a guarded resource (enter a room, open a gate, get a printout from a shared printer, …) will get authenticated by a credentials reader located near PoA using one of supported "Authentication Methods" (usually by reading an ID of RFID card). The system then logs the access attempt, verifies whether the user should have access to this resource at this time, and if so, performs an action required to grant access.

1.1 Previous work on Project Deadlock

Project Deadlock was created to provide an open-source and open-hardware alternative to existing commercial systems. It was designed to be suitable for use on Faculty of Mathematics, Physics and Computer Science. Requirements and design principles of Project Deadlock were developed collaboratively by Adam Dej, Kamila Součková and other members of ŠVT.[1] In 2016, thesis "Design and implementation of an RFID access control system" [17] was published. It describes a design and implementation of a Server component and a communication protocol utilized by Project Deadlock.

1.2 Design principles and system requirements

Design principles and system requirements of Project Deadlock were previously described in [17]. I will summarize them here in short:

- **Trustworthiness**: "[The system] must allow access when and only when it is supposed to"

[1] Študentský vývojový tím, https://svt.fmph.uniba.sk/ (Stutent Development Team)

  - **Reliability**: Partial outages of the system must not adversely affect rest of the system nor cause loss of access logs.
  - **Security**: The system must not allow illegitimate access nor access to data which would allow unauthorized duplication of access credentials.

- **Practicality**

  - **Extensibility**: The system design must allow for iterative development and be dynamically adaptable to required changes in the future.
  - **Ease of development**: The system will be maintained by Project Deadlock Community,[2] consisting mostly of volunteers with limited time resources. Entry cost for a developer should therefore be minimal.
  - **Ease of use**: Without sacrificing generality, the system should be simple and convenient to use.
  - **Ease of deployment and maintenance**: Simple deployment with minimal overhead.
  - **Availability**: The system should be inexpensive to manufacture and deploy, its components should be easily replaceable if currently used models are discontinued.

1.3 Project Deadlock architecture

The requirement of system extensibility is best fulfilled by a modular architecture. Therefore, the Project Deadlock consists of 3 types of components:

- **Server**: A software managing the whole system, keeping access rules and access logs.
- **Controller**: An embedded device with IP network connectivity which controls a single Point of Access (for example, opens a door).
- **Reader**: A user-facing embedded device which obtains user credentials using some supported Authentication Method (for example, reads an ID of RFID card). It also provides an user interface (usually LEDs and a speaker) indicating current status of the system and of the Point of Access.

Each component type may have several different implementations. These implementations communicate via standardized physical interfaces and network protocols, and as such, may be mixed and matched to create deployment of Project Deadlock that can control multiple different types of Points of Access.

[2]Project Deadlock Community is an umbrella term for Project Deadlock contributors from various development groups, such as ŠVT, or companies.

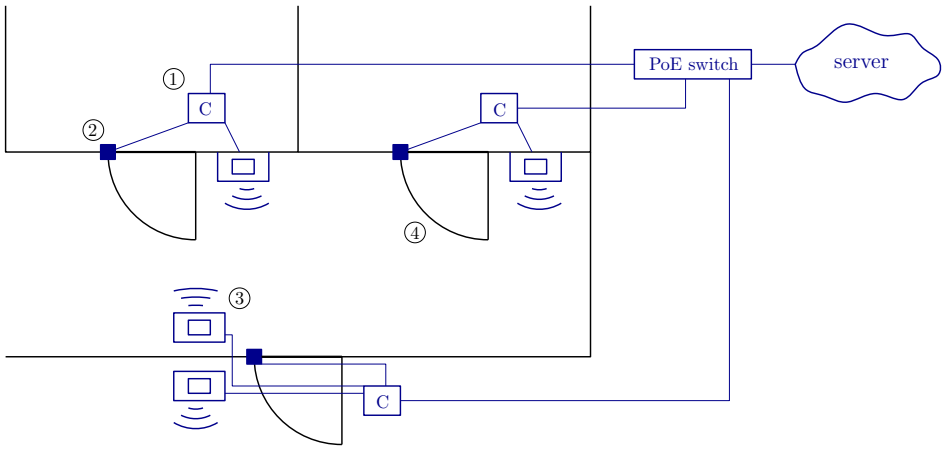

Figure 1.1: Example deployment of Project Deadlock

Figure 1.1 illustrates an example deployment of Project Deadlock. Three doors (4) are physically secured by electromagnetic locks (2). These locks are driven by Controllers (1). There is one Controller per one Point of Access. Controllers are connected using Ethernet cable, and can optionally be powered using Power over Ethernet [10] technology. They use an IP-based network to communicate with Deadlock Servers ([17] specifies an UDP protocol that supports communication with multiple servers with failovers and load-balancing). One or more Readers (3) can be connected to a single Controller. They are connected by a physical interface that is specific to Project Deadlock, but can be built from standard connectors and cables.

1.4 Current scope of the project and expected growth

The example deployment illustrated by Figure 1.1 is feasible with current components. Previous work [17] describes implementation of a Server, a communication protocol that Controllers can use to communicate with the server and front-ends to the server.

In this thesis we describe implementation of a Reader capable of reading IDs of RFID cards and of a Controller capable of controlling doors equipped with an electromagnetic lock, hold-open self-closing mechanism and “door-open” contact.

However, in the future we would like to add support for controlling different types of points of access:

- Turnstiles
- Garage doors and gates
- Shared printers (user could authenticate to the printer by reading an RFID card, and the printer will respond by printing out jobs the user previously sent to it)

Additionally, different authentication methods than reading an ID of an RFID

card are planned:

- Reading protected sections of the card (to verify that the card was not cloned)
- Using GPG encryption keys on a smart-card to sign a challenge sent by the server. This additionally requires a PIN entry method.
- Authentication using NFC-equipped Yubikey NEO hardware tokens.

To ensure that these features can be added later without the need for a major project redesign, we need to define abstraction concepts. That is the reason we have referred to guarded resources as "Points of Access" as opposed to doors. Additionally, we use term "Authentication Methods" to refer to different ways the system may authenticate the user and "UI Classes" to describe capabilities of User Interfaces present on various Readers.

1.4.1 Authentication methods. As described above, Project Deadlock must be designed to be ready to support different ways of authenticating the user. Ideally this should be achievable by a simple component swap (or even better, a firmware upgrade), without the need for a major redesign of key Project Deadlock components. Therefore we devised a concept of "Authentication Methods". Authentication Method refers to a type of authentication credential and the process of obtaining it from the user.

Currently only one Authentication Method is supported. This method obtains (4, 7 or 10 byte) ID of an ISO-14443A compliant Proximity Integrated Circuit Card[3]. This authentication method has a huge security drawback – the card ID is publicly readable[4]. Therefore, adding other, more secure authentication methods is planned in the future.

1.4.2 Deadlock Component Registry and component names. As we have described above, multiple Authentication Methods, types of Points of Access, Readers, UI Classes and Controllers may exist within the ecosystem of Project Deadlock. Different implementations of Readers and Controllers can even be considered subprojects of Project Deadlock. Since these components must work together, they must have a way of identifying each other. They must also have a common identification for the features they support.

This task will be fulfilled by a Deadlock Component Registry[5]. Deadlock Component Registry is a single source of truth repository containing machine-parsable and human-readable definitions of the following:

[3]International Student Identity Cards, used by our university, are ISO-14443A compliant and therefore this Authentication Method can read their IDs.

[4]Fast long-range ISO-14443A card readers do exist. Successful card cloning just by sitting on a bench next to a person wearing his/her badge was demonstrated in [16].

[5]This registry did not exist at the time of writing. It will be created in a near future.

- Available authentication methods, including detailed description of requirements of the particular authentication method.
- Supported types of Points of Access, including detailed description of mechanisms required to control these Points of Access.
- Controller hardware models. Each has an ID, a codename and a list of types of Points of Access it supports.
- Reader hardware models. Each has an ID, codename, UI Class, and a list of Authentication Methods it supports.
- Firmwares. Each has an ID, codename, and a list of supported hardware boards.

Each Deadlock component is assigned a numerical identification. These numerical identifications are optimal for machine communication, but may be difficult to remember for developers. Therefore each hardware component and firmware implementation also has a codename after a mineral.

Currently defined entities are:

- Authentication Method 0: Reading ID of an ISO/IEC 14443 (A) - compliant RFID cards.
- Point of Access type 0: Doors equipped with an electromagnetic lock, and optionally with "door-open" sensor and self-closing hardware with "hold-open" functionality.
- UI Class 0: User interface containing two green and two red LEDs, and a speaker. This user interface is suitable for informing the user about state of Point of Access type 0.
- Reader hardware model 0: "Fluocerite" board. A board equipped with an STM32 MCU, MFRC-522 RFID reader module. It features UI Class 0 and is capable of Authentication Method 0. Support for other RFID authentication methods can be added by the means of a firmware upgrade.
- Controller hardware model 1: "Neptunite" board. A board containing NanoPI-Duo minicomputer with Allwinner H2+ processor. It can be powered using Power over Ethernet and contains interfaces for two Readers. It can control Point of Access type 0.
- Firmware type 0: "Chibaite". A ChibiOS based firmware suitable for use on low-power MCUs, featuring an extensible RFID stack. Compatible with the "Fluocerite" board.
- Firmware type 1: "Pyroxene". A Python-based firmware running on a minimalistic Linux distribution, suitable for use with Controllers. Compatible with the "Neptunite" board.

Chapter 2

Hardware design of embedded devices

2.1 Common hardware design elements

2.1.1 Custom boards. Section 1.2 describes several requirements that have driven design decisions in embedded devices. Most notably the requirements of Reliability, cost-effectivity and ease of development. There are several ways to design a hardware for embedded device. We will analyze advantages and disadvantages of three most common approaches:

One possible approach is physically assembling components of a development platform. This approach is most common for one-of-a-kind prototypes and in early development. Example of this is buying development platform similar to Arduino and several extension "shields" that can extend its capabilities. It is very easy to develop and therefore a perfect way to create a prototype. However, no hardware cost optimization is possible, and the result is a stack of components that was never designed to be reliable over a long term.

Another possible approach is creating a custom hardware by assembling electronic components on a universal board. This approach a little bit more difficult, but since one can design his/her own circuit and choose own components, cost optimization is possible. Unfortunately the resulting design can't be assembled by a machine, which means the more costly manual assembly is required (if reliable result is desired). Also not all components are available packages that are physically usable with universal boards.

Therefore we have chosen to design completely custom boards. These can be professionally manufactured for a low cost. The drawback is that development of these boards is difficult and time-consuming. However, it allows us to design and cost-optimize our own circuit, and allows us to use larger selection of components. The

result is low-cost, space-efficient and reliable embedded device. We believe that these advantages offset disadvantages caused more difficult development.

2.1.2 Manual assembly and machine-based assembly. We have two options when it comes to assembly of our own embedded devices. Machine-based assembly is cheaper in larger volumes, provides more consistent results and can handle more component types. On the other hand, manual assembly is more effective in low production volumes, especially for prototypes, however several component package types can't be assembled this way. Taking the above into consideration, we have decided to limit our component selection to components that can be soldered manually, however optimize design for machine-based assembly (particularly by preferring SMD[1] components).

2.1.3 Extensibility. In the project overview we have described an architecture where a single embedded component type may have several different implementations. This is important especially for embedded components. Controlling different Points of Access may require different hardware (for example opening an electromagnetic lock on a single door may be accomplished by a single relay, however controlling garage door requires the ability to drive a motor in two directions, measuring its speed and processing inputs from several sensors to ensure safety of the user). Similarly, authenticating user using different authentication methods requires different Reader hardware (reading 13.56 MHz cards vs. reading fingerprints).

However there are multiple ways to implement this extensibility, two of which will be discussed:

- Physical "shields"
- One device for one purpose

The first method, physical shields, works by creating a base platform with only essential features (power distribution, processing unit) and then providing a physical connector (usually a pin header) with GPIOs and other communication interfaces. Another board may then contain additional hardware (in case of Controllers for example relays to activate electromagnetic locks) which is then connected to this on-board pin header. In certain situations the GPIOs and communication interfaces may physically pass through the shield so that another shield can be stacked on top, allowing for stacks of shields and creation of a custom mix of required hardware.

On the other hand "one device for one purpose" is a much simpler method. A single hardware board is designed to serve a single function. The only extensibility of the board can be achieved using firmware upgrades. If another set of feature is needed, another board must be designed.

[1] Surface Mount Device, a type of electronic component package that can be soldered on a surface of a board without drilling holes.

It seems obvious that the shield method will provide a better extensibility. However, there are other requirements to consider. Section 1.2 states that the system should be reliable. Introducing a extra pin header through which a critical component will be connected means introducing another point of failure. Moreover, in order for a system to be reliable it must be testable. Having to perform (and possibly automate) testing of a large number of possible different configurations will complicate the quality assurance process and ultimately lead to a less reliable product. The shield-based approach would also complicate design and increase the cost of the base platform since the designer of the base platform would have to anticipate what possible shields may be needed in the future and add required hardware to the base platform accordingly.

For this reason we have decided not to pursue shield-based extensibility.

2.2 Reader hardware implementation "Fluocerite"

2.2.1 Board features. For the Reader, we have designed a simple, small 5cm by 5cm board. This boards contains RJ12 connector, which implements a physical interface described in section 4.1. This interface provides both power and data communication lines. The power is transformed to stable 3.3 Volts by an on-board step-down converter. Data communication lines are terminated by MAX-3232 integrated circuit, which converts RS-232 voltage levels used on data lines to 3.3 volt levels, usable by on-board microcontroller. The MCU is an 32-bit low-cost low-power STM32F072 that provides us with 64kB of flash memory and 16 kB of RAM. It drives an external RFID module based on MFRC-522. This module is connected over a pin header, and contains MFRC522 IC and an antenna. On the user-interface front this board features two red and two green LEDs, and a speaker (connected over an amplifier with volume control). It can also contain circuitry required for MiniUSB connector.

2.2.2 Previous design and its drawbacks. First generation of the Reader was primarily oriented at cost reduction (of both material and development costs). In this iteration, it was powered by a tiny 8 bit MCU (Atmel ATTiny841) with 8 kB of flash memory and 512 bytes of RAM. This was not enough to hold a proper implementation of ISO/IEC 14443 (A) card reading algorithm, nor driver of the used RFID module, so this functionality was implemented in the Controller. The Reader implemented a simple communication protocol, could drive LEDs and speakers, and forwarded other commands to the RFID module.

Reader and Controller were interconnected using a single 4-wire cable. 2 wires carried 3.3V of voltage required to power the reader, the other two were used for UART link with 3.3 volt signaling levels.

This solution was implemented and deployed. Unfortunately it suffered from

several issues:

- Length of cable interconnecting Controller and Reader was limited. The Reader was powered directly from the cable without any voltage conversion, so the system was sensitive to voltage drops on the cable. The current design solves this issue by rising on-cable voltage, and implementing a voltage converter on the Reader.
- Since low voltage UART was used for signaling, the induced interference on the cable had more pronounced effect on the system. This limited the baudrate and caused the need to retransmit packets more often, slowing down the communication. The current design solves this issue by using RS-232 line levels, and implementing a level-shifter on the Reader.
- Speaker on the reader was not loud enough. This is solved by inclusion of an amplifier that is powered directly from the cable voltage.

2.2.3 Further work. The current design iteration has one important issue. RFID Reader module is essentialy an extension board connected over pin header. Disadvantages of this approach are described in section 2.1.3. Additionally, this RFID module is built using low-quality components, and more than 50% of these modules can't read ISIC cards. Therefore they have to be manually modified, and retrofitted with inductors rated for higher currents.

Less important, but still frustrating, issue with this board is that due to a design oversight, UART transmit and Serial Wire Debug data I/O is connected to the same physical pin, therefore only one can be used at any given time. This meant that if a bug was found in communication library during firmware development, the developer could choose to either be able to reproduce the issue, or attach a debugger, but not at the same time.

The next iteration should not use an external module, but instead it should contain the required circuitry and antenna on the Reader board itself.

2.3 Controller hardware implementation "Neptunite"

Neptunite Controller is implemented on a 7.5cm by 7.5cm printed circuit board. It is built around a small NanoPI DUO [14] board, which contains an Allwinner H2+ CPU, 256MB onbaord RAM, 10/100M Ethernet PHY and an SD card slot. This board is powerful enough to run Linux efficiently. Physically this board is inserted to a socket in the Neptunite controller board.

We have designed the Neptunite board to be powered either by an external 50 Volt adapter, or using Power over Ethernet. PoE circuitry on board is compatible

with both passive and active IEEE 802.3-2012 power sourcing equipment. Up to two Readers can be connected to this board (intended for Readers on both sides of the door).

It is equipped with 2 relays, that can be wired to act as switches, or can supply either fixed 12 Volts or any voltage between 3 Volts and 10 Volts via dedicated onboard voltage regulator. Intended use is the control of a single Type 0 Point of Access, where one relay drives the electromagnetic lock, and the second relay activates hold-open function of door closing mechanism. An input for "door-open" contact sensor is also available.

2.3.1 Further work. No cost optimization was yet performed for this board. PoE circuitry provides 50 volts, and an expensive off-the-shelf step-down converter is used to convert it to 12 Volts. This converter can be redesigned to use components that are still sufficient and ultimately cheaper. The PCB layout is inefficient, resulting in a board that is larger than necessary, increasing costs further. Onboard dedicated user-adjustable lock power supply is absolutely unnecessary for the job. It can be completely replaced by a single PWM controller. All these issues will be fixed in a next iteration of the board.

Chapter 3

Reader firmware "Chibaite"

The Chibaite firmware is intended to be used on "Fluocerite" Reader hardware (described in section 2.2) or on other Readers that use a low-range (and low-cost) microcontrollers[1], that are not capable of running mainstream operating systems such as Linux. This firmware must be fast, stable and execute efficiently on a low-power MCU. We have therefore decided to write this firmware in C.

To mitigate disadvantages of writing a C firmware directly for a bare-metal device (such as portability), we have decided to use ChibiOS [4]. ChibiOS is a development environment for embedded applications that can freely be used for open-source projects[2]. It contains an RTOS (ChibiOS/RT), which provides threading capabilities, synchronization primitives and other useful programming constructs, and HAL, which simplifies development and facilitates firmware porting to different hardware boards. This way, this firmware can be used on more Reader hardware models, if necessary.

3.1 Components and their interaction

The Chibaite firmware is composed of several Tasks and one Master Task. Each task runs in its separate thread and does only one thing. For example, we have implemented a task that reads RFID cards and another task that updates the UI. These tasks do not communicate directly with each other. Instead, they only communicate with the Master Task. The Master Task implements business logic of the Reader firmware. For example, when the Master Task receives a callback from RFID Reading task that a card ID has been obtained, it will instruct Communication task to send that ID to the Controller.

This threaded approach was chosen because it simplifies event dispatch and

[1] At the time of writing, this firmware, compiled using `gcc` and `-O2` takes up approximately 32kB of flash space, which is roughly half of what is available on the Fluocerite board.

[2] ChibiOS is dual-licensed. It can be used either under GNU GPLv2 license for open-source projects, or under a paid license for commercial projects.

state management of the device. Instead of managing a complex global state, each task isolates its state from rest of the system and manages it on its own. Each thread can then implement simple event handling loop in a way that is best suited for the task. This overally simplifies the development.

Since every Task (including the Master Task) runs in a different thread, a form of inter-thread communication is required. Each task provides a thread-safe API which the Master Task may call whenever it needs. This allows Master Task to send instructions to individual Tasks. For the other communication direction we have chosen a different approach. A Task specifies a set of callback functions the Master Task must implement if it wants to use the given Task (in a thread-safe way). When the Task is being initialized, the Master Task passes pointers to its implementation of these callback functions as argument to initialization function of the Task.

This Task callback approach, similar to a "dependency injection" concept, was chosen because it simplifies testing of the firmware. This way, each task can be easily separated from the system and tested as a separate unit. We discuss testing embedded devices in more detail in Chapter 6.

Functionality of this firmware is provided by 3 simple tasks:

- **UI task** is responsible for the user interface, displaying LED flash sequences and driving the speaker.
- **CardID task** is responsible for polling for and reading IDs of RFID cards. It provides an implementation for Authentication Method 0.
- **Comm task** handles communication with the Controller.

3.1.1 Watchdog. The multithreaded nature of this firmware also presents some issues. Since each task runs as a separate thread, it may lock up / enter an endless loop without impairing the rest of the system. If the *CardID* task locks up, the Controller will not notice anything abnormal, however the Reader will cease to function from perspective of the user. It is therefore necessary to implement a watchdog which would watch over all tasks (including the Master Task), and reset the Reader if something like this happens.

Restarting the whole Reader seems like a crude solution, however it is effective in getting the Reader to a known state. The Reader can start up less than half second. The communication protocol described in Section 4.2.1 provides a connection-oriented service, and if the Reader gets reset, the link drops. This in turn notifies the Controller that it needs to reinitialize the Reader.

The implemented watchdog mechanism utilizes the hardware watchdog built in the MCU. The watchdog in this MCU is essentially just a timer, which when runs out, resets the MCU. So the firmware has to perpetually reset this timer. Resetting the watchdog is the responsibility of the Master Task. Each task, including a Master Task,

generates heartbeats. Code that generates these heartbeats is written in such a way, that it depends on correct functioning of the monitored task. For example, the *CardID* task polls for a card in a loop. After each loop, it generates a heartbeat. If the task gets stuck when reading a card, it will not be able to generate this heartbeat. If Task uses more than one thread (as is the case for the *Comm* task, each thread generates its own heartbeats).

Those heartbeats are implemented as callbacks to the Master Task. The Master Task internally keeps a vector of bits (Heartbeat Vector), where each bit represents a single thread. When a task calls its Heartbeat callback, the Master Task sets its bit in the Heartbeat Vector. Then, when the Master Task itself generates a heartbeat, it checks the Heartbeat Vector, and if bits representing all threads present on the system are set, it resets the watchdog timer, and clears all bits in the Heartbeat Vector. Therefore if a task continually misses its heartbeats, its bit will not be set and the watchdog timer will not be reset which will cause restart of the Reader.

3.2 Previous component design and its drawbacks

The architecture described in section 3.1 is a direct result of an attempt to implement firmware with overly-complicated design, which retained only advantageous elements. This section describes this design and explains its drawbacks.

The proposed firmware consisted of several modules. Each module implemented a single well-defined task, ran in its own thread and communicated with other modules using message passing. This way, each module could be individually monitored for correct functioning and restarted if necessary. Facilities required for this were provided either by the underlying operating system (ChibiOS, in our case) or part of the firmware called "Common Deadlock Services".

Common Deadlock Services were supposed to implement two notable features:

- CMA, Central Message Allocator. This system was responsible for allocating messages that modules use to communicate with each other.
- Component Supervisor, which was tasked with starting, watching over and if needed restarting firmware modules.

There are however huge issues with this approach. Implementing "Common Deadlock Services" as proposed would require essentially creating a garbage-collected programming environment in plain C on embedded hardware, where memory access issues are very hard to debug. Any advantages in firmware stability the monitoring and task restarting would bring would be negated by the potential for bugs present in the Common Deadlock Services themselves. Additionally firmware modules would have to continue functioning properly even if any other modules they depend on are

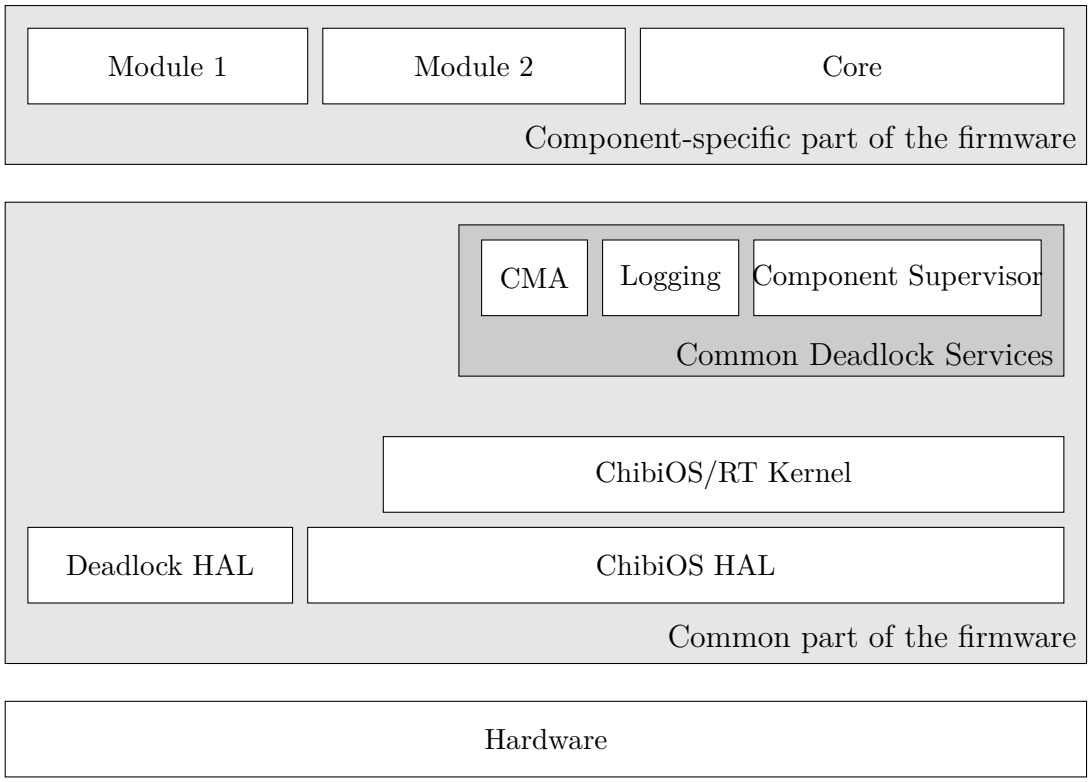

Figure 3.1: Proposed, but unused, embedded firmware architecture

restarted at any moment. These two factors would have created failure modes that are very hard to predict and difficult to debug.

3.3 RFID Stack

The Chibaite firmware was primarily intended to implement Authentication Methods that require communication with an RFID card (most notably Authentication Method 0, reading an ID of ISO/IEC 14443 (A) compliant PICC). For this, a robust RFID stack was required.

Our requirements for the RFID stack were:

- Implementation of a proper Anticollision sequence, as described by ISO/IEC 14443-3 [11]
- Either the ability to "select" a PICC and communicate with it, or clean and structured codebase that would allow us to add this feature
- Usable with ChibiOS/HAL and optimized for embedded devices
- Abstraction between the code handling communication with the card (PICC), and the code handling communication with the RFID hardware module (PCD).

At first, we tried to find an open-source implementation that is already done. Fortunately, easy access for hobbyists to PCD module we have decided to use, and to

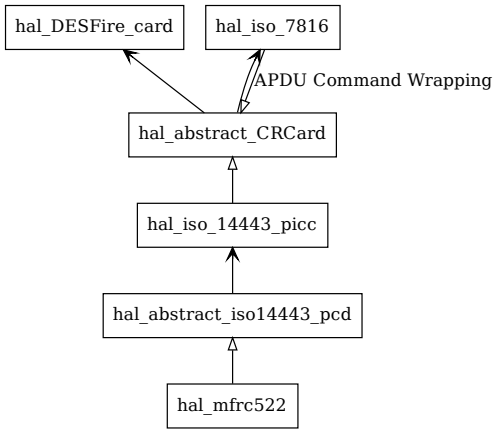

Figure 3.2: RFID stack architecture

microcontroller development platforms, such as Arduino, meant that many different implementations were available. Unfortunately, most of them were either specific to Arduino, had ISO/IEC 14443-3 anticollision sequence implemented incorrectly, had low quality code base or were mixing low-level PCD drivers with PICC activation logic. In the end we found no suitable candidate.

Therefore we have implemented our own RFID stack that fulfills all requirements we have specified above. Most importantly, our RFID stack provides a proper abstraction layer between PCD driver, and PICC communication logic, which increases portability of this firmware to boards featuring different PCD hardware (section 2.2.3 describes why such a change may be necessary). Figure 3.2 illustrates the architecture of our RFID stack.

3.3.1 Components of our RFID stack.

3.3.1.1 Proximity Coupling Devices

Proximity Coupling Device is a hardware module physically capable of communicating with a card (PICC). In order to be able to support several hardware modules, we have defined an abstract interface `hal_abstract_iso14443_pcd`.

Communication between the PCD and the PICC consists of sending and receiving frames. The frames are transmitted in pairs, PCD to PICC followed by PICC to PCD. Part A of the standard defines 3 different types of frames:

- Short frame: transmits 7 bits.
- Standard frame: Used for data exchange and can transmit several bytes with parity.

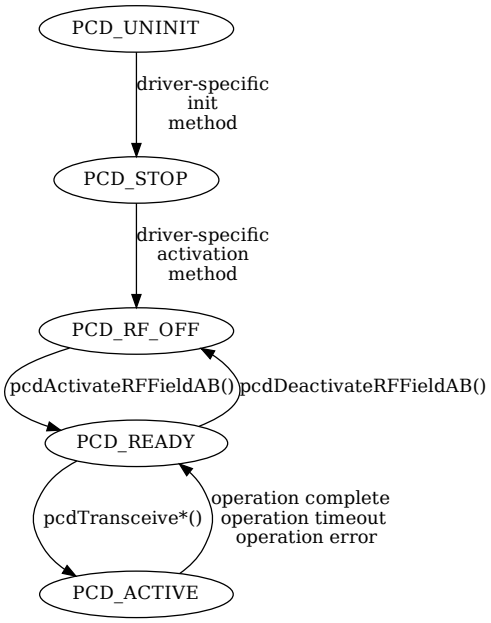

Figure 3.3: States and state transitions of a PCD driver

- Bit oriented anticollision frame: 7 byte long frame split into two parts. First part is transmitted by the PCD, second part is added by the PICC. It is used during bit-oriented anticollision loop.

ISO/IEC 14443-3 specifies different communication methods (different modulation type / index, different encoding, several speeds) for part A and B. However, not all PCDs support all modes, therefore this interface provides a mechanism for PCD feature querying.

PCDs also usually support a number of extended features, not covered by the ISO/IEC 14443 standard. For example, the MFRC522 is able to perform a Mifare authentication using its crypto unit, or a self-test. Upper layers which know how to use these extended features should have access to them, but they should not clutter the main API. We have provided a mechanism for accessing these, outside of the main API.

This interface is implemented by a MFRC-522 driver (`hal_mfrc522`), that can drive PCD present on the Fluocerite board.

3.3.1.2 Proximity Integrated Circuit Cards

This part of RFID stack uses the PCD interface to implement drivers for activating and communicating with RFID cards themselves. We have defined an interface called `hal_abstract_CRCard`. It defines an abstract interface used to communicate with an Integrated Circuit Card (either with contacts or contactless) using request-response

frames.

If we want to communicate with an ISO/IEC 14443 PICC, we must first activate ("select") it. In order to activate a card, we must know its ID. Part 3 of ISO/IEC 14443 standard defines a procedure that can be used to get IDs of all cards present in an RF field [11]. This procedure is called "Anticollision cascade".

The standard defines three possible lengths of IDs of PICCs (4, 7 and 10 bytes). Only 4 bytes can be obtained in a single anticollision loop, therefore to get the whole PICC ID, we may need up to three loops (called "cascade levels"). We start with a first cascade level, and eventually obtain 4 bytes of PICCs ID. Then we try to issue a "select" command. The card will respond to this command, and indicate whether its ID is complete or not. If it is not, we discard the last received byte, and proceed with the next cascade level.

Getting a part of an ID in a single cascade level works by exchanging "Anticollision frames" with PICCs (see section 3.3.1.1). An anticollision frame contains the following 7 bytes:

- CMD: the command (this depends on the cascade level)
- NVB: Number of Valid Bits
- UID[0-3]: Part of the UID CLn (UID in Cascade Level n)
- BCC: Checksum, all previous bytes XORed

This frame may be split into two parts anywhere after the second byte and before the last byte. First part is transmitted by the PCD. It includes a part of ID it already knows[3]. The second part is transmitted simultaneously by all PICCs with IDs that start with the prefix transmitted by the PCD. All PICCs transmit at the same speed, and therefore transmit a bit at a given position at the same time. If there are multiple PICCs in the field, they will have different IDs and therefore inevitably a situation will occur, where two PICCs will transmit two different bits at the same time. A PCD can detect this situation at physical layer. This is called a "collision".

When a collision occurs we store received valid bits before the collision, and remember the collision position. Then we transmit the first part of the anticollision frame again, with all known bits and the collided bit set to 1. Only PICCs with matching ID prefixes will respond again with the second part of the frame. There may be a collision again so we repeat the procedure in a loop. When we know the full ID we append a CRC to the full Anticollision frame. This is called the SELECT command, which will activate the PICC (or tell us to proceed to the next cascade). After that, we go back to the last collision position where we previously set the collided bit to 1. This time we set it to 0 and repeat the whole process. This way we can get UIDs of

[3]In the first iteration the PCD does not know any ID bits and therefore NVB is zero. In that case, all cards in the RF field will respond.

all cards in the RF field. If the number of found cards exceeds a constant the user can pick (`max_cards`) then we simply won't mark the collision position and we will set the collided bit to "1". That way we ignore all cards with lower IDs.

This effectively means that the user does not have to remove a card from his/her wallet, since the Reader is capable of reading IDs of all cards present there.

We have implemented the process described above in component `hal_iso_14443_picc`. This component also detects all cards in the RF field, and constructs a `CRCard` object for each one. This object contains ID of the card and is directly usable by Authentication Method 0, or by other layers of this RFID stack that may wish to communicate with the card further.

3.4 Bootloader

Bootloader, in the context of embedded devices, is a dedicated small piece of software that runs before the main firmware starts. It is most commonly used to update the main firmware. Currently, Chibaite firmware does not implement its own bootloader, but relies on one that is built-in to the STM32 MCU. This has the advantage that no custom implementation is required. Unfortunately, the built-in bootloader does not support any form of access control when reading flash memory, nor signature verification when upgrading the firmware. It can only be completely disabled.

3.5 Further work

Even though this firmware serves its purpose well, there are several weak spots that should be improved. The RFID stack requires more stability testing. Currently, in certain situations, it can freeze or not detect a collision properly. Custom bootloader should be implemented and the MCU should be configured so that no other side-channel can read the main flash. This would make it safe to store access keys if flash memory of the MCU. Additionally, the custom bootloader should allow firmware upgrades only if they contain a valid cryptographic signature, so that firmware of this device couldn't be hijacked for nefarious purposes. After that, advanced Authentication Methods should be implemented. Currently only Authentication Method 0 is supported, and it is known to be not secure.

Chapter 4

Reader/Controller interconnection

In this chapter we describe in detail physical interfaces and all layers of communication protocols used to connect Readers to Controllers. Then we describe the `libdeadcom` component, a universally usable, heavily documented and automatically tested library that implements these protocols.

4.1 Physical layer

Readers and Controller are physically interconnected using a 6-wire cable terminated with RJ12 (6P6C) connectors. Maximum length of such interconnecting cable is 15 meters. This single cable carries power, data lines and two other auxiliary signal lines:

Pin	Function
1	RST
2	Vcc
3	reader RxD (controller TxD)
4	reader TxD (controller RxD)
5	GND
6	Boot

Power is provided by the Controller, with absolute minimum voltage of 4 Volts, and absolute maximum voltage of 15 Volts. Nominally the Controller should choose to provide either 5 Volts or 12 Volts. In any case, it must be able to provide at least 3 Watts of power.

RxD and TxD lines are RS-232 communication lines with voltage ranges as mandated by TIA/EIA-232-F standard [18]. This allows for a communication that is stable, fast and is easily implemented by a single level shifting chip and almost any cheap embedded microcontroller. Initial communication over this interface uses the

following mode:

Baudrate	Data bits	Stop bits	Parity
38 400	8	1	Even

After the contact between Controller and the Reader is established, a new set of communication parameters may be negotiated[1].

Two auxiliary signals `RST` and `Boot` are also available. Logical signaling level on those pins is 3.3 Volts. `RST` is active LOW, idle HIGH. `Boot` is idle LOW, active HIGH. LOW pulse on `RST` line will generate a reset of the Reader. If `Boot` signal is active during power-up or reset, the Reader enters its Bootloader mode (see section 3.4).

4.2 Data link layer

Reader and Controller must be able to reliably exchange messages. If, for example, the Reader is attempting to send message "Authentication Method 0: ID Obtained", this message must either be delivered reliably or error must be detected (and user must be notified of this error). Therefore it is clear that reliability will have to be ensured at some layer. We have decided to solve this problem on the Link Layer. Since communication between Reader and Controller consists of exchange of short messages rather than streams, this layer must provide facilities for datagram transport. The system must also know if the communication link between Reader and Controller is currently functional, therefore this layer must provide a connection-oriented service.

Providing reliable connection-oriented datagram communication over unreliable byte-oriented link is a problem that is already solved by a High-level Data Link Control (HDLC) protocol [12]. We were especially interested in HDLC "Asynchronous Balanced Mode" of operation, which would allow any station to initiate a connection and communication with the other. Unfortunately, we were unable to find an open-source implementation suitable for use in embedded devices. We have therefore decided to create our own implementation of such library.

4.2.1 Deadcom Layer 2 (`dcl2`) library. Deadcom Layer 2 library implements a protocol that fulfills the above requirements. Internally it uses a protocol inspired by HDLC operating in Asynchronous Balance Mode, however simplified to simplify its implementation and maintenance. This simplification lowered the required development time, however as a result of this decision, `dcl2` library itself is not compatible with HDLC. It also has a limitation that a station must wait for acknowledgment of a frame it sent before it is allowed to send another frame.

[1] Although none of the current devices require nor make use of this feature.

Internally this library utilizes a modified version of open-source library `yahdlc` [20]. This library claimed to provide support for encapsulating byte buffers to HDLC frames, including application of control-octet transparency, and also extracting frame contents from arbitrary byte streams. We have discovered and fixed several issues in this library and extended it to support frame types that we required. In order to minimize code footprint, we have also removed unneeded APIs[2]. Since these changes are not backwards-compatible, no attempt to upstream these changes was made. We have incorporated our `yahdlc` fork directly into `dcl2` (in compliance with the MIT license `yahdlc` is released under).

This library is intended to be used in multithreaded environments, and takes advantage of this fact for the implementation of its procedures of operation. For example, the function to transmit data to the other station blocks the calling thread (by waiting on a condition variable) until either reception confirmation from the other station is received or the operation fails. Before unblocking the calling thread with error status, the transmit function attempts to retransmit the message several times. In the meantime, responses are processed in a "Receive thread". This thread is responsible for extracting received messages, automatically acknowledging frames that were already received (this can happen if ACK frame got lost) or processing acknowledgments from the other station and signaling condition variable the sender thread is waiting on. This design simplifies both usage and implementation of this library since we can avoid status callbacks that may occur at any time.

To function properly, this library requires an environment that provides two synchronization primitives: mutexes and condition variables. Functions dealing with initializing and using these objects are passed as function pointers to `dcl2` initialization function and stored as a part of `dcl2` object. It is up to the user to provide these "glue" functions for the platform of his choice[3]. This "object-oriented" approach is advantageous if a Controller wants to communicate with multiple readers at once. However the real advantage of this approach is testability, since this allow for existence of multiple `dcl2` objects simultaneously with different implementations of these (mocked, in the case of unit tests) "glue" functions. This is used heavily by unit tests of this library[4].

4.2.1.1 `dcl2` API

From user's perspective this library provides a simple API. Function `dcInit` initializes an object representing a Deadcom L2 link. This is the function that takes "glue"

[2]Full list of changes made to the `yahdlc` library is detailed in license comment of `yahdlc` files. Those can be found in `libdeadcom` repository.

[3]Requirements on these functions are as minimal as possible and it is known that they can be implemented on ChibiOS, on any system with `pthread` environment or on any system with working `threading` Python module.

[4]These tests run automatically on Gitlab's Continuous Integration system.

function pointers as described above. Function `dcConnect` attempts to create a connection. It will block until the connection is established or the operation times out. Function `dcDisconnect` closes the connection immediately. Messages can be transmitted using function `dcSendMessage`, which blocks until the other station acknowledges the reception or the operation times out. Function `dcProcessData` processes a part of received byte streams, automatically sending responses where necessary and unpacking received message to internal buffer. Function `dcGetReceivedMsg` can then be used to obtain complete received message from the internal buffer. Complete documentation of this library is automatically generated from sources using Doxygen and Sphinx. It is available in in `libdeadcom` repository.

4.2.1.2 Helper libraries

We have also created library `helper-pthreads` to simplify usage of this library on systems where `pthread` library is available. This library implements threading "glue" functions using `pthread`. It is heavily utilized by integration tests of the `dcl2` library itself.

Additionally, we have created library `leaky-pipe` as an extension of open-source project `pipe` [15]. Project `pipe` provides a plain-C thread-safe implementation of a FIFO queue. `leaky-pipe` adds configurable deterministic unreliability to the `pipe`. It can be configured to lose, corrupt or add bytes to the communication, either in a specific sequence or in pseudo-random way. This library can be used to simulate a thread-safe unreliable byte-oriented link, and is used in integration tests of `dcl2`. (Of course this library also has its own unit tests).

4.2.1.3 Python 3 bindings

We have decided to implement Controller software in Python 3 (see Chapter 5 for details). So in order to use `dcl2` library from Python, we have created an extension for CPython. This extension consists of two parts: Low-level module `_dcl2`, written using Python C API and high-level module `dcl2`.

The low-level `_dcl2` module provides a way to call C API of the `dcl2` library directly from Python. It takes care of Python type and object creation and converts arguments and return values between Python types and C types. It also implements dummy "glue" functions required by `dcl2` that are just invocations of callback functions the user can implement in Python[5].

This module is in turn used by `dcl2` Python module. `dcl2` Python module provides a high-level Pythonic API. Internally it utilizes `threading` module to implement

[5]This is the only time the design of `dcl2` library caused implementation difficulties. It turned out that implementing a Python extension that requires callbacks to Python code from a multithreaded C environment is not that easy. However once implemented the result turned out to be worth it.

"glue" functions and uses them to initialize the low-level `_dcl2` module. Additionally it creates a "receive thread"[6] that periodically reads bytes from a file-like object the user has provided and calls function to process these bytes. This greatly simplifies usage of this library.

To demonstrate simplicity of this solution, let us compare usage of C implementation with the usage of these Python bindings. To use this `dcl2` C library (without `helper-pthreads`) the user has to:

- Use platform-specific way to initialize a serial port
- Implement 6 threading glue functions and one transmit function that uses that serial port
- Create a structure with function pointers to these functions
- Initialize the library
- Create a receive thread that calls `dcProcessData` periodically when data were received over the serial port

The same thing can be accomplished by the following Python code:

```
import dcl2
import serial

s = serial.Serial("/dev/ttyS1", baudrate=38400, parity=serial.PARITY_EVEN)
link = dcl2.DeadcomL2(s)
```

4.3 Application layer

Application layer of Reader/Controller communication protocols facilitates exchange of commands and event notifications between Reader and Controller. These devices communicate by exchanging Controller-Reader Protocol Messages (CRPMs for short). CRPMs contain data encoded in RFC 7049 (Concise Binary Object Representation) [2] format. CBOR is a format with data model similar to JSON, however designed to be easily implemented with minimal code footprint, and produce compact binary-encoded messages. These messages can be encoded and decoded without knowing the schema in advance, allowing for easy extensibility of this protocol in the future.

Even though the schema definition is not necessary for encoding/decoding CBOR messages, we have decided to specify one since schema specification with comments explaining details can serve both as a protocol documentation for developers

[6] Since we are using CPython we are limited by CPython's Global Interpreter Lock, which means that CPython can't run more than one thread of Python code at any given time. This is however not a problem for us, since our task is I/O-bound.

and a formal protocol definition usable in automated tests. We have used CBOR Data Definition Language (CDDL) [9] to describe the schema[7]. This schema specification format can be also used by utility based on Ruby gem `cddl` to check if a given string of bytes contains schematically-correct message, and to generate schematically correct random data[8].

4.3.1 Controller-Reader Protocol Messages. The Controller-Reader protocol consists of several CRPM types. Each CRPM type is assigned a number and also allowed direction (whether this message can be only sent by Controller, only sent by Reader or if it can be sent by both). Each CRPM message is a CBOR map containing one entry. The key is the CRPM Type number and the value is CRPM payload. Expressed in CDDL:

```
crpm = {
    0 => heartbeat //
    1 => sysQueryRequest //
    2 => sysQueryResponse //
    3 => activateAuthMethod //
    4 => rdrFailure //
    5 => uiUpdate //
    6 => am0PiccUidObtained
}
```

4.3.1.1 Heartbeat CRPM

This is a dummy CRPM for checking whether the other station can receive and acknowledge a message. No response to this message is expected on application layer. This protocol is intended to be used on top of Deadcom Layer 2 protocol, which implements acknowledgments, and if the other station is unable to acknowledge this frame, we will be notified of transmission failure.

This CRPM has no payload and can be sent by both Reader and Controller. In CDDL:

```
heartbeat = nil
```

[7]Unfortunately, at the time of writing, CDDL is only an Internet-Draft, not an RFC standard. We have used the latest available draft which is due to expire in August 2018. Once it reaches RFC status the schema definition will be thoroughly reviewed and fixed if specification has changed.

[8]Similarly to CDDL Internet-Draft, `cddl` Ruby gem is a work-in-progress, and even though it can use schema defined by us to verify existing messages, it does not yet support some of the CDDL constructs we have used and can't generate random schematically-correct data.

4.3.1.2 System Query Request CRPM

This is a no-payload CRPM that can be sent by Controller to Reader to identify it. The Reader must reply with "System Query Response".

```
sysQueryRequest = nil
```

4.3.1.3 System Query Response CRPM

Reader uses this CRPM to respond to "System Query Request" received from a Controller. The payload is Reader Hardware model number, Firmware type number, hardware revision, hardware serial number and software version. Hardware model number and Firmware type number must be registered in Deadlock Component Registry.

During the design of this CRPM we were faced with a dilemma: should the Reader report its model and firmware, or just a list of authentication methods it supports and type of UI it features? The latter is in the end what the Controller is interested in. Additionally it would allow us to swap Reader for another with the same set of features, that the Controller may not yet know about, which would allow for extensibility and iterative system upgrades. However, we believe that reliability is more important, and this approach would allow using Readers with Controller that have never been tested together. And even though it should work in theory, in practice this may lead to issues that are discovered (if we are lucky) only in production environment.

Therefore the Controller can use these values to find out whether it supports this particular Reader model and its Firmware, and which UI and Authentication Methods this Reader supports.

```
sysQueryResponse = [1*1 sysQueryResponseGroup]
sysQueryResponseGroup = (
    swType:   uint .size(2),
    hwModel:  uint .size(2),
    hwRev:    uint .size(1),
    devSN:    text .size(25),
    swMajor:  uint .size(1),
    swMinor:  uint .size(1)
)
```

4.3.1.4 Activate Auth Method CRPM

Controller can use this CRPM type to instruct Reader to enable one or more Authentication Methods. Available authentication methods are enumerated in the Deadlock Component Registry.

```
activateAuthMethod = [ 1*10 &authMethod ]

authMethod = (
    am0PiccUid: 0
)
```

4.3.1.5 Reader Failure CRPM

Reader can use this CRPM to inform Controller of its own failure. Payload is a human-readable text. The Controller should therefore log this failure for review by the operator.

```
rdrFailure = text .size(1..200)
```

4.3.1.6 UI Update CRPM

The Controller can use this CRPM to instruct the Reader to update its UI status to make user aware of changes that have occurred on the Point of Access. Available UI classes are enumerated in the Deadlock Component Registry. Currently only one UI class is supported.

```
uiUpdate = &uiClass0State

uiClass0State = (
    doorClosed: 0,
    IDAcceptedDoorUnlocked: 1,
    IDRejected: 2,
    doorPermanentlyUnlocked: 3,
    doorPermanentlyLocked: 4,
    systemFailure: 5,
    doorOpenTooLong: 6
)
```

4.3.1.7 Authentication Method 0: PICC UID obtained

The Reader can use this CRPM to inform the Controller that Authentication Method 0 has succeeded in obtaining IDs of one or more ISO/IEC 14443 (A) Proximity Integrated Circuit Cards. These IDs may be 4, 7 or 10 bytes long.

```
am0PiccUidObtained = [1*10 piccUid]

piccUid = bytes .size(4) / bytes .size(7) / bytes .size(10)
```

Chapter 5

Controller firmware “Pyroxene”

The Pyroxene firmware is intended to be used on the “Neptunite” Controller hardware, described in section 2.3, or on any other board with enough resources to run Linux effectively. The software itself is written in Python 3.

5.1 Embedded Linux distribution

Since we have decided to base this firmware on Linux, we needed to find an appropriate Linux distribution. The distribution must be small, because storage space could be limited (Fluocerite board supports SD cards, however industrial SD cards[1] are still expensive). Many exploits for poorly secured embedded Linux systems are currently in the wild[2], so the distribution must not only be secure by default, but it should also be difficult to make it insecure by accidental misconfiguration.

Flexible, but rolling release distributions such as Arch Linux, which were used in early prototypes, were immediately rejected. After some deliberation, we have also decided against use of conventional distributions, such as Debian. These distributions are designed and optimized for use on desktop and server systems. Their ARM ports do not receive the same level of thorough testing as their x86_64 counterparts. They contain a lot of software necessary for its function but ultimately unnecessary for embedded Controller, so it only increases the attack surface.

Instead, we have decided to use tool `buildroot` [1] to create our own minimal Linux system. This system contains only packages that are essential to its operation:

- Linux kernel 4.16
- `musl`, a fast and lightweight standard library
- Busybox, which implements basic utilities and an init system
- Python 3 interpreter

[1] Use of consumer-level SD cards is strongly discouraged due to their limited durability.
[2] Probably most commonly known is malware Mirai, which was detected in August 2016.

Initially we wished to use an LTS release of the Linux kernel. However, Neptunite board is based on Allwinner H2+ CPU, and support for Ethernet MAC controller present in this CPU was included in mainline version of Linux only in release 4.15. As soon as LTS release of post-4.15 Linux kernel will exist this firmware will migrate to it.

The Linux kernel is configured to be as stripped-down as possible. Support for loadable modules is disabled, since we know exactly which drivers will be needed beforehand, and it also simplifies the boot process. Only drivers that are really necessary are included in the kernel. Busybox is stripped down in a similar way.

The system build process generates two images: a compressed kernel image, and a compressed squashfs image of root filesystem. This means that root filesystem is not only mounted read-only, it is impossible to modify it. This further limits the attack surface, since it is impossible to silently add a malicious program/file. When the firmware needs to be upgraded, it is upgraded as a whole with another image. A separate read-write data partition is provided for network configuration files, Python implementation configuration, access logs and access database. No code should be executable from this partition.

Currently, the squashfs root filesystem takes less than 8MB of space, and this distribution can boot in under two seconds (not including Python application starup or bootloader delays)[3].

5.1.1 Hardware abstraction. Similarly to Chibaite (Reader) firmware, this firmware should also be able to support multiple different hardware boards. Linux provides several different mechanisms which we can use for this purpose.

Configuration of Linux kernel can be separated into several files, using KConfig fragments. Therefore we can define a default kernel configuration, configured for needs of this distribution, and a hardware configuration fragment for each board we wish to support.

Additionally, we can use "Device Trees" to inform the kernel about different devices that are present on a given board and how are they connected. With this information, the kernel can properly configure all devices during bootup. Additionally, we can use it to specify names of various LEDs and GPIO pins that are present on the board. This way, the userspace can only ask the kernel "Activate gpio deadlock:lockrelay"[4], and the kernel will do the right thing as long as information in the Device Tree is correct (so it does not matter if the relay is physically connected to

[3]And its build is automated, using Gitlab's Continuous Integration.

[4]Surprisingly, this feature was added to Linux kernel relatively recently. This change also introduced a new way to control GPIO from the userspace (by addition of a character device), and deprecated the old and inflexible ABI for controlling GPIO using `/sysfs`. Unfortunately not many people have noticed and not many Python libraries are available for this purpose.

processor GPIO pins, or over a GPIO expander that communicates with the processor over I2C bus, as is the case for Neptunite board).

5.1.2 A/B firmware upgrades. A/B firmware upgrades work by keeping 2 copies of the system in persistent storage at the same time, and updating the one that is not running. Then, only a reboot is required to switch to the new version[5], which limits the system downtime. Additionally, if the boot fails, the system can recover by booting the old version again, which improves system reliability.

We are planning to implement this feature in Pyroxene firmware. Storage is partitioned so that it can contain 2 copies of the system. Das U-Boot is used as bootloader. U-Boot supports loading and storing persistent environment variables from a fixed storage location. These variables can then be used in boot scripts to choose which copy of the system should be loaded. U-Boot also supports incrementing a variable each time a boot is attempted. Additionally, it is possible to modify these variables from Linux. The firmware upgrade process can therefore look like this:

- Flash rootfs and kernel to partitions that belong to currently inactive system
- Modify U-Boot environment variable that points to system that should currently be active
- Reboot
- U-Boot will increment "number of attempted boots" variable, and attempt to boot the new system
- If the system boots, it should reset "number of attempted boots" variable back to zero, and copy itself to partitions occupied by the original system (so that it will not be possible to boot the old, potentially insecure version by accident).
- If the system does not boot, the "number of attempted boots" won't get reset, and if it reaches a sufficiently large value, the U-Boot will fall-back to booting the old system, which may then perform a recovery action.

5.2 Further work

Unfortunately, at the time of writing, the Pyroxene firmware is largely unfinished. Build system for the Linux distribution is mostly done, however init system and network configuration are missing, as well as business logic implementation. These will be implemented in a near future.

Pyroxene firmware should support Neptunite board, which is equipped with a PoE+ power extractor. The IEEE 802.3-2012 [10] standard mandates that a Powered

[5]This method is successfully implemented for example in Container Linux by CoreOS or in newer versions of Android.

Device must support both Hardware and Link Layer classification, and the Power Sourcing Equipment may decide which one to use. Neptunite board supports hardware classification, but Link Layer classification must be implemented in this firmware. The `lldp` deamon may be used for this purpose.

Chapter 6

Testing

This chapter describes methods for testing software of embedded devices and other components of this project, especially the Chibaite firmware. Software testing is essential to ensuring that the code is of high quality. It is also beneficial in many other ways. However, testing code for embedded devices is not easy since in production environment the code runs on an embedded hardware and there is no option to force specific error-conditions upon that hardware.

In case of Pyroxene firmware, which is written in Python, unit testing is easy. The situation is more complicated for Chibaite firmware written in C. There are 3 feasible options. Some of them have advantages, unfortunately all of them have disadvantages.

- Compile the code for PC and run tests locally
- Run tests on an emulator of the given embedded platform
- Run tests on physical embedded hardware

Compiling code for the PC and running these tests locally is very fast compared to other options. However, only hardware-independent pieces of code can be tested in this way. Moreover, since different compiler is used to compile the unit tests these tests won't catch compiler-introduced bugs (which is not unheard of in embedded development). However, logical error is still a logical error no matter the platform or compiler, so these tests can test business-logic very well. Although it is possible to test higher-layers of device drivers by mocking the low-level device interface and emulating the hardware, it is pointless since it is way harder to write a flawless hardware emulator (bug-for-bug compatible with the real hardware) than it is to write a solid driver for it.

Running code on an emulator has the advantage of using the same compiler for both tests and production code. MCU registers may be changed at will and error conditions introduced. Unfortunately, we have not found suitable emulator for our platform.

Running tests on physical embedded device is the most difficult approach. Physical hardware can't be forced to deterministically create a specific failure mode through software when debugging drivers. Collecting test results is also difficult. Unfortunatelly, this is often the only applicable choice.

6.1 Chosen testing strategy

In Chibaite firmware, we have decided not to write automated tests for the lowest Hardware Abstraction Layer (two device drivers we have created) for aforementioned reasons. These will be tested manually. Other drivers that we have used are implemented in ChibiOS HAL [5], which is rigorously tested by its author.

The Chibaite firmware has modular design. Each module runs in its own thread and is designed so that it can be separated from other modules and tested on its own (see Section 3.1). These modules will be unit-tested by cross-compiling them for a host PC.

Tests for Python components of Pyroxene firmware are written in a way that is specific to Python.

Unit tests and integration tests of `libdeadcom` are run also by compiling these components for host PC.

6.2 Unit tests

Unit tests are built on the Unity [19] unit testing framework in combination with the Fake Function Framework [8].

6.2.1 Technical details of unit-testing C code with mocking. Mocking is a process of replacing an object or function with dummy version mimicking real objects / functions for testing purposes. If one `.c` file represents one unit then unit-testing is relatively easy: compile the file under test, compile the test file, let the test file define mock functions, link them. Test file will call a function from the file under test, it will call some library function which is provided by the mock in the test file.

Usually a single `.c` file is a single module consisting of several units, and a unit is represented by one function. Functions in a `.c` file may use other functions from that file, so in order to write an unit test for given function from the `.c` file, other functions from this file may need to be mocked. Even worse, these functions may be static (for optimalization reasons). There are several solutions to this problem:

- Split the `.c` file into multiple files so that each file is an unit. Advantage is that building and writing tests is easy. Also it may encourage better and more

modular design. Disadvantages are that you need to produce more `.c` files and the code may be harder to write. Also static functions can't be used.

- You can use weak references (through some preprocessor magic like custom defined `testable` macro or by `objcopy --weaken`). You can then provide your own implementation of custom symbols in the test file and linker will replace weak references in the file under test. Advantage is that writing tests is easy, disadvantage is that you may need magic macros or a bit more complicated build process. You still need preprocessor magic to solve `static` functions, as compiler may do what it wants with `static` functions (such as inline them or change their calling convention).
- You can manipulate the GOT table at runtime and force the running program to use your function instead [13]. Building tests is easy, writing them is hard and the whole process is not elegant and unreliable. And it still won't solve the problem with `static` functions.

We will do the following: where possible and logical we will split the code to multiple `.c` files. It is quite possible that the split will not be needed in most cases, and when it is needed the file should anyway be logically splitted. Additionally the file will be processed with `objcopy --weaken`, so if it **really** is not logical to split the file under test it is still possible to mock internal functions.

The last problem to solve are `static` functions defined in the same file and `static inline` functions defined and present in headers of the OS. These functions are written in this way for performance reasons, and examples include functions for entering/exiting OS critical zones.

Static functions will be solved by custom preprocessor macro. It is not the most elegant solution, but it will work reliably. The problem with `static` or `static inline` functions can't be solved in this way, since they are defined in OS headers, which are chain-included from one include which can't be ommited. This can be solved by including `testable.h` header **after** all system headers. This header can be specific for each test and can use preprocessor to rename used functions. Those renamed functions can then be easily mocked. This is also very ugly solution, however, it will be required only in rare cases.

6.2.2 Used mocking framework. We use Unity [19] unit testing framework. This framework is also used by CMock [6] mocking framework, from the same creators, so it would be logical to use it. That, however, turned out to be a bit problematic.

CMock parses header files included by the file under test and produces a mock file for all functions it finds. Advantage is that this process is fully automatic. Disadvantage is that CMock treats one file as one unit and makes it impossible to mock functions inside that file.

The other problem is the build system. The mocks should be generated automatically and tests should be built automatically as well. Officially recommended option is to use Ceedling [3] as a build system. However, this is not universally applicable to this project, especially to the Chibaite firmware which is based on ChibiOS, build system of which is based on Makefiles which include other ChibiOS-specific makefiles from within ChibiOS folder structure. Ceedling is based on Ruby's Rake, and can't build ChibiOS based projects.

CMock itself supports Makefiles, however it is quite inconvenient to use it. It works by generating a new Makefile using a Ruby script. This script makes assumptions about the project which don't hold true for the Chibaite firmware (e.g. flat source code structure). Moreover, official documentation mentions that it is required to run `make` twice for this to work. The last option is to use ruby build system `rake`. This was actually feasible, since you are free to utilize internal CMock objects (which is also written in Ruby), and the example was easily modified to build the tests with our code structure. Rake was invoked from the Makefile with proper enviroment variables (because they are known only during execution of `make`). However, this solution is not elegant since it mixes two different build systems.

All that would be still quite acceptable. However, the biggest problem is that CMock doesn't work with ChibiOS. CMock works by parsing included header files and finding function definitions in them. However, it is not able (nor it is intention of authors to make it able) to preprocess that file. In case of ChibiOS the user includes only `ch.h`. Other parts of the system are included by this header, and are dependent on the architecture. The preprocessor is heavily utilized and the resulting include is dependent on compile-time definitions defined in ChibiOS Makefiles. This makes perfect sense for embedded projects, as it minimizes footprint of the compiled code. However, it also means that in order to reliably generate mocks the files have to be preprocessed. Official advice on this was to run the header through the C preprocessor. After we've done that the CMock crashed with a parsing error on the resulting file. In the end we've decided that although CMock is a decent framework it is not applicable for our use case.

We have decided to use the Fake Function Framework [8] instead. It's usage, as opposed to CMock, is trivial: `#include "fff.h"`. However now we have to do mocks manually. But this is actually a great advantage, since we can mock what we want when we want to. Usually we don't have to mock that many functions anyway, and all it takes to mock a function is `FAKE_VOID_FUNC(halInit);`. And if we forget to mock something the linker notifies us. And we can use the weak-reference trick and mock functions from the same file that the function under test comes from.

6.2.3 Building and running tests. Unity includes an example Makefile which can be used to build and run unit tests. It needs to be modified to be usable in our case, because:

- It presumes a flat source / test folder structure
- It doesn't automatically generate test runners
- It doesn't automatically weaken references of object file under test
- It supports only one test for each file

So we have written our own makefile for running tests. For explanation of this file please refer to the Chibaite documentation [7] or review Makefiles in `libdeadcom` repository.

Conclusion

We started this thesis by introducing the Project Deadlock and describing its architecture in chapter 1.

Then, in chapter 2 we have described decisions that were the main drivers of hardware design. Later we describe design of two hardware boards for our embedded components, Fluocerite (Reader implementation) and Neptunite (Controller implementation). We have manufactured and tested the Fluocerite board. Neptunite board was assembled as a prototype, and final boards are being manufactured at the time of writing. Additionally, further work such as integrating PCD module to Fluocerite board itself, and cost-optimization of Neptunite board was described.

Chapter 3 described a Chibaite firmware, written in C and based on ChibiOS. We have described our previous mistakes, and what lessons we have learned. Then we have described implementation of the firmware itself. As a notable part of this firmware, we have also implemented a clean and modular RFID stack capable of reading identification cards. Then we summarized further work this firmware requires, notably implementation of more secure authentication methods.

Design and implementation of all communication layers utilized by Readers and Controllers was described in chapter 4. First we described the physical link. Then we described requirements of Data Link Layer, and how our implementation fulfills them. Several helper libraries for testing this implementation were also developed, as well as bindings of this library with Python. Lastly, we have specified an Application layer protocol using CBOR Data Definition Language.

In chapter 5 we have described design of Pyroxene firmware for Controller. Unfortunately, we have managed to implement only a small part of underlying Linux distribution. However, we have described architecture and our future plans for this fimware.

We have concluded with chapter 6, which described various methods used to test different components of this project.

Appendix A: Source code and documentation

Project Deadlock is under continuous iterative development. Source code, hardware schematics and documentation of various components of this project can be found at https://gitlab.com/project-deadlock/.

Project homepage (with documentation and development guides) will be hosted at https://project-deadlock.io/ in the near future.

Attached: CD with source code and hardware schematics of Project Deadlock components described in this thesis.

References

[1] Buildroot - Making Embedded Linux Easy: 2018. *https://buildroot.org/*. Accessed: 2018-05-14.

[2] CBOR: 2017. *http://cbor.io/*. Accessed: 2017-03-20.

[3] Ceedling: 2017. *http://www.throwtheswitch.org/ceedling/*. Accessed: 2017-01-27.

[4] ChibiOS: 2017. *http://www.chibios.org/dokuwiki/doku.php*. Accessed: 2017-01-27.

[5] ChibiOS - HAL: 2017. *http://www.chibios.org/dokuwiki/doku.php?id=chibios:product:hal:start*. Accessed: 2017-01-27.

[6] CMock: 2017. *http://www.throwtheswitch.org/cmock/*. Accessed: 2017-01-27.

[7] Deadlock reader documentation: 2017. *http://deadlock-reader-sw.readthedocs.io/en/latest/*. Accessed: 2017-01-27.

[8] Fake function framework: 2017. *https://github.com/meekrosoft/fff*. Accessed: 2017-01-27.

[9] H. Birkholz, C. Vigano, C. Bormann 2018. Concise data definition language (cddl): A notational convention to express cbor data structure. Internet-Draft draft-ietf-cbor-cddl-02 (Feb. 2018).

[10] IEEE 2009. Data Terminal Equipment (DTE) Power via the Media Dependent Interface (MDI) Enhancements. 802.3at Amendment 3 (Sep. 2009).

[11] ISO/IEC 1999. Identification cards - Contactless integrated circuit(s) cards - Proximity cards - Part 3: Initialization and anticollision. ISO/IEC 14443-3 (Jun. 1999).

[12] ISO/IEC 2002. Information technology — Telecommunications and information exchange between systems — High-level data link control (HDLC) procedures. ISO/IEC 13239:2002(E) (Jul. 2002).

[13] Mimick: 2017. *https://github.com/diacritic/Mimick*. Accessed: 2017-01-27.

[14] NanoPi Duo: 2017. *http://wiki.friendlyarm.com/wiki/index.php/NanoPi_Duo*.

Accessed: 2018-05-14.

[15] Pipe.c: 2017. *https://github.com/cgaebel/pipe*. Accessed: 2018-03-11.

[16] Real time RFID Cloning in the Field: 2017. *https://www.youtube.com/watch?v=kUduHIygbY8*. Accessed: 2018-05-10.

[17] Součková, K. 2017. Design and implementation of an rfid access control system. (2017).

[18] Telecommunications Industry Association (TIA) 1997. Interface between data terminal equipment and data circuit-terminating equipment employing serial binary data interchange. TIA/EIA-232-F (Oct. 1997).

[19] Unity - throw the switch: 2017. *http://www.throwtheswitch.org/unity/*. Accessed: 2017-01-27.

[20] yahdlc - Yet Another HDLC: 2017. *https://github.com/bang-olufsen/yahdlc*. Accessed: 2018-03-11.