Smart Door Lock

A first prototype of a networked power lock controller with an NFC interface

RAFID KARIM
and
HAIDARA AL-FAKHRI

KTH Information and Communication Technology

Degree project in
Communication Systems
First level, 15.0 HEC
Stockholm, Sweden

Smart Door Lock

A first prototype of a networked power lock controller with an NFC interface

Rafid Karim
and
Haidara Al-Fakhri

2013-12-01

Kandidatexamensarbete

Examiner and academic adviser
Professor Gerald Q. Maguire Jr.

School of Information and Communication Technology (ICT)
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

Most major cell phone manufacturers have been releasing cell phones equipped with Near Field Communication (***NFC***). At the same time there is also increasing use of mobile payments and user verification with the use of the NFC technology. These trends indicate both the increasing popularity and great potential for increased use of NFC in today's society. As a result NFC has a huge potential to simplify our everyday tasks, ranging from paying for items to accessing our office or home.

In this context we will focus on using NFC together with a Power over Ethernet (***PoE***) powered circuit board and NFC reader to realize a simple system for granting access to open a locked door. One of the purposes of this realization is to explore what services can be realized when such a system is connected to the home/building network and connected to the Internet. A second purpose is to learn how to use network attached devices, as the concept of the Internet of Things is considered by many to be a driving force in the next generation Internet. This project uses very in expensive and low power hardware, as the number of devices is potentially very large and thus in order to minimize the technology's impact on the environment we must consider how to minimize the power used – while maintaining the desired user functionality.

This bachelor's thesis project made it possible for a PoE powered circuit board containing a MSP430 microcontroller to work along with a NFC reader, which was connected through the Serial Peripheral Interface (***SPI***).

We hope that the end result of this project will lead to a simpler life by exploiting this increasingly ubiquitous technology. For example, a homeowner could send a one-time key to a repair person who is coming to fix their sink. Similarly a homeowner could send a key to their neighbor which is valid for two weeks so that their neighbor could come into their home to water the plants while they are away on vacation. Another example is lending your apartment key to a friend while you are out of town.

Keywords: Near Field Communication, Power over Ethernet, secure access, Internet of Things, Serial Peripheral Interface, MSP430

Sammanfattning

Det blir allt vanligare med närfältskommunikation (NFC) i dagens samhälle, mobiltelefons-tillverkarna börjar utveckla nya telefoner med NFC teknik inbyggd, samtidigt som användningen av NFC ökat.

Det sker även en utveckling inom mobila betalningar och användar-verifiering med användning av NFC, då NFC förenklar detta.

Med detta sagt kommer vi att arbeta med detta i detta kandidatexamens-arbete där vi fokuserar på NFC samt Power over Ethernet som använder MSP430 chippet som kärna. Med dessa enheter kombinerade kommer en enkel rörelse med ett NFC kort över en NFC läsare som sedan skall ge åtkomst till en låst dörr. Detta i större kombination med en Internetuppkoppling kunna ge ägaren möjligheten att kunna skicka ut dörrnycklar till andra användare.

I detta kandidatexamensarbete gjorde vi det möjligt för ett PoE kretskort bestående av ett MPS430 mikroprocessor att samarbeta med en NFC läsare genom SPI protokollet.

Genom att utveckla detta projekt hoppas vi att vårt slutresultat leder till en enklare delning av nycklar med hjälp av denna teknologi.

Nyckelord: Närfältskommunikation, Ström via nätverk, Säker åtkomst, Sakernas Internet, Synkron Seriekommunikation, MSP430

Table of contents

List of Figures

List of Tables

List of acronyms and abbreviations

BSL	Bootstrap Loader
DCO	Digitally-Controlled Oscillator
DHCP	Dynamic Host Configuration Protocol
FET	Flash Emulation Tool
GSM	Global System for Mobile
GPS	Global Positioning Systems
ISO	International Organization for Standards
JTAG	Joint Test Action Group
KB	Kilo Byte
MCU	Micro Controller Unit
MSP	Mixed-Signal Processor
NDEF	NFC Data Exchange Format
NFC	Near Field Communication
PCB	Printed Circuit Board
PD	Power Device
PoE	Power over Ethernet
PSE	Power Sourcing Equipment
RFID	Radio Frequency Identification
SBW	Spy-Bi-Wire
SPE	Source Power Equipment
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UDP	User Datagram Protocol
UID	Unique Identifier Number
UICC	Universal Integrated Circuit Card
UMTS	Universal Mobile Telecommunications System
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
Wi-Fi	Wireless Fidelity

1 Introduction

In this chapter we will give a short overview and explanation of this thesis project.

1.1 General introduction to the area

In the last several years, smartphones have become more powerful and have been designed to be used as pocket-sized personal computers. As different designers and smartphone manufacturer try to "win" market share they are constantly adding new features and improving the performance of their different models of smartphones. This has enabled other companies to develop new tools and services which utilizes smartphones. Today, Near Field Communication (NFC) is provided in many smartphones. This has enabled applications to verify the user's location* and in some cases their identity by using a smartphone. This technique in combination with others can be used in various applications and services.

Computer communication systems and the Internet are playing an important role in our everyday environment. Today almost any device (workstation, television, lamp, etc.) can be connected to the Internet. Increasingly smartphones are constantly connected to the Internet over third and fourth generation networks. This network connectivity will play an important part in our project.

As technology develops, the demands for new products and services to make our life more efficient also grow. Consider the simple use case of lending your apartment key to a friend while you are out of town. This could be made easier by use of NFC and smartphones in combination with the Internet communication, as you can simply issue you friend a digital certificate which he or she can present to your door via their smartphone's NFC interface. To realize this use case we will use a combination of technologies to prototype a *Smart Door lock.*

1.2 Problem definition

To develop our smart door lock there are some sub-problems we need to solve. First we need to study two basic technologies: NFC technology (as used in smartphones) and a network attached door lock. We will combine these technologies to develop our smart door lock. Based upon our study of NFC we must create an application that can run on a smartphone† to respond to the NFC reader when it is queried. We need a corresponding application running in either the network attached door lock or in the cloud to query the smartphone via NFC. Given the NFC communication between the reader and the smartphone an application running in either the network attached door lock or in the cloud will determine whether the door should be unlocked or not.

While we have some basic experience with microcontrollers and some knowledge of computer communication systems, we did not yet have any knowledge of NFC technology. Combining these different technologies in one project should take our knowledge to the next level. Our first step in doing this is to connect a microcontroller to the Internet, and then connect a NFC reader to this microcontroller. Note that one of the other areas that we want to explore is the use of Power over Ethernet (PoE) technology, so that we do not need a separate connection from our microcontroller to the building's power mains.

1.3 Goal

The main goal of this project is to develop a smart door lock system where an administrator or owner of an apartment or building can manage and send "digital keys" to other persons in order to allow them to access an apartment, building, or specific room in a building by using their NFC equipped smartphone or a NFC smartcard.

* See for example the use case of a security guard in the recent thesis by Thi Van Anh Pham, Security of NFC applications [1]

†In order to achieve to this during our time of the project, the smartphones required to run in Card Emulation mode, something that we didn't have the possibility to use, but was later introduced in Android KitKat 4.4

The project can be divided into two parts: the first part concerning the NFC module in the phone and the NFC reader connected to the microcontroller and the second part which concerns the communication between the microcontroller and a cloud service. We have adopted a cloud deployment approach based upon the energy savings shown by Goce Talaganov in his master's thesis [2] and because of the flexibility that a cloud based service will offer.

Figure 1-1 shows the overall system. As shown in this figure a NCF reader will be placed outside the door and the microcontroller will be placed inside the building where it cannot be modified or access by someone outside the door. The microcontroller will be connected to a server via a Power over Ethernet (PoE) capable switch. This network connection provides power to the microcontroller, NFC reader, and electric strike plate (or motor to turn the latch). Further information about PoE technology will be given in section 2.5 on page 7. The system will be managed through a website where the administrator of a lock can create electronic keys and manage the lock. Note that these keys can be designed to work only during a specific time or even be a one-time-only key.

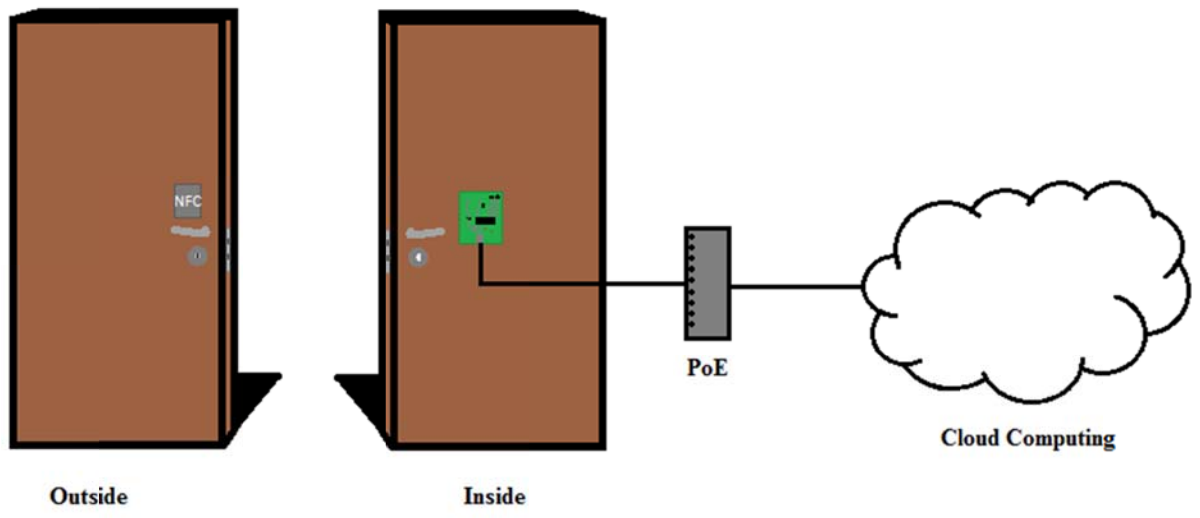


Figure 1-1: An overview on how the smart door locks system works (Note that the figure does not show the connection to the electric strike plate or latch.)

The system must be connected to the actual lock on the door; this can be done in different ways depending on the type of lock installed on the door. The most common lock used in houses is a dead bolt; these locks use a metal bolt that slides into the door jamb which is controlled by a key on the outside and a latch on the inside (see Figure 1-2). For these kinds of locks, were there is no electronics managing the lock, hence we have to install a motor, servo, or a solenoid that can switch the latch on the inside of the door to lock or unlock the door, this will require some type of sensor that can tell the microcontroller if the door is locked or unlocked, these sensors can be buttons that will be pushed if the latch is in the locked- or unlocked-mode.

There are also electronic door locks that already have a build in motor or solenoid which unlocks the door when it gets a signal from a controller unit. This controller can be attached to a card reader or a key pad for PIN-codes. For these kinds of locks the microcontroller of the smart door lock system can be connected to the lock mechanism as a second controller unit. However how the physical lock should actually function and what kind of locks to use is not the focus of this project.

The detailed sub goals to be achieved are listed at the start of Chapter 3.

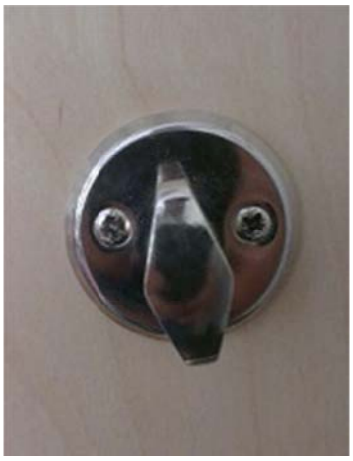


Figure 1-2: **A latch to mechanically control the door lock on a door**

1.4 Structure of the thesis

This thesis is divided into five chapters. Chapter 1 gives an overview of what the project is about. Chapter 2 will give the reader basic background material so that the reader can understand the concepts that will be subsequently used in this project. This chapter will also summarize some of the related work relevant to this project. Chapter 3 covers the methods used in the project to achieve our goals which contains a range of both software and hardware tools. Described in chapter 4 is how we tested our prototype to see if it fulfilled our purposes with the project. Since we did not accomplish all of our goals, chapter 5 reviews our conclusion and describes what we have left undone in addition to suggesting what could be done in future work to build upon this project. The final chapter also covers what we would have done differently in the project if we were to do it again.

2 Background

In this chapter we will briefly describe the embedded platform that we will use and we will introduce some of the concepts that are useful to understand the rest of this thesis. This chapter will also describe some of the related work that is relevant to this thesis project.

2.1 What have others already done?

Some previous research has been done that is related to our project. We will summarize these related projects in this section. This is particularly the case for our basic platform, as we will re-use the microcontroller and PoE network circuit board developed by earlier master's thesis students at this department. Following this we will introduce some of the related work done regarding door locks, access control, and cloud based services.

2.1.1 Exploiting Wireless Sensors

The master's thesis[3] by Albert López and Francisco Javier Sánchez concerned sniffing wireless sensor traffic in order to collect this sensor data and use it for multiple purposes. They worked with sensors in the 868 MHz band. They designed and created a motherboard with a TI MSP430 microcontroller as the core of their gateway. Since the MSP430 is a very low power consumption chip it was ideal for use with PoE. They utilized a Microchip ENC28J60 network interface. This network interface was connected to the microcontroller via a serial peripheral interface (***SPI***). The ENC28J60 Microchip offers dual port random access memory for sending and receiving data packets, as this network interface provides the buffering needed for packets being sent and received, there was no need for external memory. In order to supply power to this board they used PoE technology. In their project they also used an SPI interface to connect a daughterboard with a radio transceiver for the 760 to 928 MHz band.

2.1.2 White space sensor platform

Javier Lara Peinado in his thesis project [4] took the sensor platform developed by López and Sánchez and added a new boot program to provide network based booting, configuration of the device via the dynamic host configuration protocol (DHCP), and installation of application software via the trivial file transfer protocol (TFTP)*. Using this new software base he implemented a white space sensor platform that sent its measurement results to a central server via UDP. White spaces can be shortly described as licensed frequencies that are not used all the time.

2.1.3 Fixing the PoE and building a white space sensor grid

Julia Alba Tormo Peiró in her thesis project [5], extended the work of Peinado and corrected the problem with respect to limited power of the PoE circuit of the board developed by López and Sánchez. We will make use of this modification to the board and the additional boards which she made in our experiments.

2.1.4 Lockitron and other commercial lock company efforts

The company Lockitron has developed a similar *smartlock* that is placed over your current interior latch for a deadbolt lock. The Lockitron product does not use the PoE technology, but rather it uses a battery for its power. Also the Lockitron product uses Wi-Fi for its network communication and can only work with specific deadbolt locks, which makes it less universal.

Assa Abloy has also created a smartlock solution for door locks. Assa Abloy's Seos product uses NFC for locking and unlocking doors [6]. Yale locks has also developed an NFC lock system, it uses the mobile lock platform that Assa Abloy developed [7].

* See section 2.3.

Vingcard Elsafe is another company that is marketing an NFC-compatible lock. At this time the product is primarily for hotel rooms. Because of its product design, this product seems invisible to the user [8].

Telcred has implemented a service that allows access to facilities with the help of digital tickets. By using these digital keys in conjunction with NFC technology they have realized a highly flexible solution for visitors to the EIT ICT Labs Centre in Stockholm [9]. (One floor below where we are carrying out our project.) After we finished the laboratory work for out project, Telcred installed three of their lock systems in doors near the laboratory where we did our project.

2.2 NFC

NFC is a contactless communication technology. Today NFC is used in higher end smartphones and tablets[10, 11]. NFC uses radio-frequency identification (RFID) techniques to communicate with another NFC device. The two devices should not be more than 10 cm from each other (a theoretical limit) and their separation should be less than 4 cm for stable communication[12]. The most common uses of NFC today are for identification and tickets. Mobile payments using NFC[13, 14] is another popular topic because of its speed and security. Given that Visa (the world leader in electronic payment) has now teamed up with Samsung (the company which sold the most phones 2013) to enable smartphones with NFC to be used as a credit or debit card [15], it seems that the future of NFC is bright.

2.2.1 Using an NFC reader

NFC operates at 13.56 MHz and it uses RFID technology for its communication. There are mainly two sorts of NFC devices: a passive NFC device and an active device that is always connected to a power source. The passive device has no internal power source; therefore it dependents on the electromagnetic field produced by the NFC reader. The active device generates an electromagnetic field that powers the passive device it wants to communicate with.

Two active NFC devices can exchange data with each other using peer-to-peer mode. Peer-to-peer mode is standardized in International Organization for Standardization (ISO) 18092 [1]. An NFC reader can also function in Card Emulation mode.

2.2.2 Security of NFC applications

Every NFC-enabled application requires appropriate security. The type of security is related to what kind of function the application provides. For example, a service that performs a monetary transaction must be highly secure (at least proportionately to the maximum value of transaction or transactions that can be performed).

In contrast, simple services such as receiving (with the help of using your NFC smartphone) discount “coupons” as you enter a shop entrance would require very little or almost no security at all. The case of an NFC-enabled restaurant menu is explained in detail in the recent master’s thesis of Thi Van Anh Pham [1].

2.2.3 Secure elements

A secure element (***SE***) is a platform were an application can be installed, managed, personalized, and executed securely. An SE is a combination of hardware, software interfaces, and protocols that allows secure storage and usage of credentials for payments, authentication, and other services[16]. SEs can be categorized into three types:

- An **embedded SE** is a non-removable SEs that can manage business- and personal information in a safe way. Embedded SEs is installed in the unit at manufacturing.
- A **smartcard for mobile phone terminals** in GSM- and UMTS-band (***UICC***[*]) is a genetic, standardized, physical, and logical platform for smart card applications. A UICC is used by

[*] Universal Integrated Circuit Card

telephone companies to include a USIM (3G SIM card) application on the card to verify a user for a 3G network.

- A **micro SD-card** (µSD-card) is a memory card with an embedded chip that is used as a SE. There are some SD-cards with embedded NFC antennas.

The form of SE used by cellular operators or payment services in conjunction with a NFC enabled phone is decided by the companies that are involved [17]. Companies have selected each of the three types of SEs described above in order to achieve their specific goals.

2.2.4 Security and Privacy of NFC applications

As mentioned earlier, the security of NFC applications is very dependent upon what kind of service the application provides. As a result there are different approaches to privacy in NFC applications. Privacy in NFC applications is almost entirely reliant on the design of the application. For example consider the matter of default settings; for privacy reasons the NFC communication capability should be inactive when a user's smartphone is locked or inactive. However, this is not always the default setting for every smartphone. Additionally every NFC application should inform a user about what action it is going to take when it is being used. For example, when an application is going to open a web browser or when the application will send a text message. An advantage of NFC from the point of view of transaction privacy is that NFC is independent of Global Positioning System (GPS) activation in the phone and can even be independent of the cellular network. These features provide the user with greater anonymity and increase the user's privacy.

2.3 TFTP

The Trivial File Transfer Protocol (TFTP) is a very simple protocol used to transfer files between network devices. TFTP was developed in the late 1970s, but stabilized in the 1980s[18]. TFTP is commonly used to transfer configuration and boot files to hardware that lack persistent memory or disk space, thus a device need not have a disk or stable storage for more than a network boot loader. As a result TFTP is widely used for upgrading or restoring firmware in routers.

TFTP uses the UDP protocol for its transfers. Typically it uses UDP port 69, but it is also possible to configure a TFTP server so it uses another UDP port number.

Since the board that we will be using supports TFTP, we have adopted this as a means for booting up the device, installing its software, and configuring the hardware. This means that we can easily install new software, without needing to use an EEPROM programmer or other similar device. However, it does mean that we need to implement a Dynamic Host Configuration Protocol (DHCP) server and a TFTP server for our experiments.

2.4 Dynamic Host Configuration Protocol server

We will use a DHCP server to supply an IP address to each of the network attached microcontroller circuit boards based upon a MAC address that we have configured into the network boot loader. The DHCP server will also provide the board with the IP address of the TFTP server and the name of the file that this board is to load via TFTP. The DHCP configuration file that we have used is shown in Appendix A.

2.5 PoE

Power over Ethernet (PoE) enables Ethernet cables to transfer both data and electrical power to devices. PoE can theoretically deliver up to a maximum of 15W of DC power. In practice the maximum available power is about 12.95W because of the losses in the cables. PoE was standardized in the IEEE 802.3af standard [19].

2.5.1 Advantages of PoE

USB does not always provide sufficient power required for some types of hardware to function. Additionally, in a smart door lock scenario the likely distances will be greater than those supported by USB. USB can provide 5W of power over a cable with a maximum length of 5m, while PoE devices using CAT5 cable and provide 12.95W of power to devices 100m away[20], CAT5e cables can provide 30W of power.

2.5.2 Disadvantages of PoE

The main disadvantage of implementing the PoE interface is that not all network switches supports the PoE interface, those switches that supports the PoE is usually more expensive. If there is no switch that implements the PoE interface available, there is PoE injectors to buy to add power to the PoE interface.

Also because the network cable is providing power there might be a concern of the cables heating up, however, this is not an issue because the power limitation at 30W for CAT5e cables is well under the safety margin from the cable bundle heating up as the CAT5e cables have a lower resistance drop [21].

2.5.3 Alternatives to PoE

Power might be provided to the device via an alternate means, such as a separate cable from an AC to DC power converter. However, in this case there is a need to run a separate power cable to the device (in addition to the cable used for communication) and there is the need to locate the AC to DC power converter at an electrical outlet. Using an AC to DC converter connected to mains power likely increases the cost, as an outlet may needs to be installed near the door or there is the cost of additional cabling.

Another alternative power source for the device is a battery or supercapacitor. This alternative avoids the need to externally power the device at the cost of adding a battery or supercapacitor. Additionally, this complicates the system as there now needs to be some means to replace or recharge this power source. There may also be a problem about whether the lock should open when it runs out of power or whether it should remain locked. The later alternative may introduce the need for another means of opening the lock if there is no power, while the former may eliminate the security that was offered by the lock.

3 Method

In this chapter we will explain our goals and what we have to do to reach our goals. There are many steps and new things we have to learn before building *the Smart door lock* system.

In order to realize *the Smart Door Lock* is a system we need to achieve these goals:

- Connect the microprocessor circuit board to a network,
- Make the microprocessor download its application at boot time,
- Connect a NFC reader to this microprocessor,
- Create an application for a smartphone that can send messages to the NFC reader,
- Create custom UDP packets to be sent and received by this microprocessor,
- Connect sensors to this microprocessor,
- Connect and control a servo motor connected to this microprocessor, and
- Set up a web server and a homepage to control this microprocessor.

All of these sub goals have to be achieved and the different elements of the system have to work together properly. For example: the NFC reader should send data to the microprocessor which the microprocessor will forward as data inside a UDP datagram to our webserver. We will also have to do the same encapsulation of data from the sensors and in order to receive commands from the web server we will decapsulate the commands received within a UDP datagram to lock or unlock the door.

3.1 Software

In this section we will explain the software we have used to develop *the Smart door lock*.

3.1.1 Wireshark

Wireshark[*] is very popular computer program that analyzes network traffic. Since our project is partly based on sending and receiving data exchange from both the motherboard and the DHCP/TFTP server, Wireshark was a very useful tool for troubleshooting and analyzing if the packets contained what we expected and were send/received when we expected. We used Wireshark to verify if there was DHCP and TFTP activity. This was very useful when we started to send our custom made UDP packets.

3.1.2 Code Composer Studio

We have used Texas Instruments' Code Composer Studio™ (CCStudio) [22] as our integrated development environment when writing code for the microprocessor.

3.2 Hardware

In this section we will describe the hardware tools that we used to develop *the Smart door lock.*

3.2.1 Mixed signal Oscilloscope

To develop and verify the signals transmitted and received from/to the microcontroller we need to actually see the signals, to do this we used an oscilloscope. The oscilloscope that we used is a HP 54645D, now known as Agilent 54645D [23], which is a mixed signal oscilloscope with two analog inputs and 16 digital channels for mixed signal analysis.

This oscilloscope supports triggering which allowed us to easily find the signals that we want to see by selecting one signal to trigger on when this signal entered the state that we wanted to investigate.

3.2.2 Programmer

The programmer that we used to flash our code to the MSP430 microcontroller is the MSP-FET430UIF from Texas Instruments (TI) [24]. The programmer is officially referred to as a Flash

[*] www.wireshark.org

Emulation Tool (FET); this tool can be used for writing the program to the MSP430 microcontroller and controlling the microcontroller manually for debugging a program, it can even provide power to the microcontroller.

The standard FET device from TI connects via the USB interface, although there are older versions that connect via the serial-port interface. This FET can program the processor using either the Joint test action group (JTAG) or Spy-Bi-Wire (SBW) protocol through a 14 (2x7) pin connection that is available on the motherboard (section 3.3.1).

3.2.3 Arduino

The Arduino board is a microcontroller board with open open-source hardware. The specific Arduino board we used was an Arduino Uno R3 [25] based on the ATmega328 [26] microcontroller. The board also has 14 digital input/output pins which can be used for different functions. The board contains a USB connection where can also be used to power the board when connected to a computer. Of course one can also power the board other ways, as so long as the power supplied is not over 5 volts.

There are many different daughter boards (so called shields) that can be easily attached to the Arduino board through I2C and SPI communication. We used one such shield equipped with an NFC interface (this shield is decribed in section 3.3.2). The main reason why we used the Arduino Uno R3 board is because we wanted to investigate the necessary connections beyond the SPI interface between the NFC shield and the Arduino board.

3.2.4 HP ProCurve Switch 2626

For testing our system we connected our motherboard's network interface to an HP ProCurve Switch 2626 [27] switch. This is a PoE capable switch; hence it could power the system. We also connected this switch to a Dell model Optiplex GX620 [28] computer via a secondary Ethernet interface.

3.2.5 Desktop PC

A Dell model Optiplex GX620 desktop computer running *openSUSE* [29] acted as the DHCP server (see section 3.4.1) and TFTP server (see section 3.4.2). This computer also ran Wireshark – which was used to capture and observe the traffic to and from the system that we were developing. This computer could also be used to provide the services that in a real deployment of the system would be provided by a server running in a cloud.

3.3 Description of the embedded platform

In this part of the thesis we will briefly describe the embedded platform that was developed by previous master students (as described in sections 2.1.1 to 0) to function as a wireless sensor sniffer. For further details about the platform we refer the reader to the individual master's thesis indicated in these sections. The platform can be separated into a main motherboard with a SPI[*] interface for connecting a daughterboard. The SPI interface on the motherboard provides the adaptability to connect a daughterboard of the user's choice without having to modify any other part of the circuit board. In the case of the former master's thesis projects the SPI was used to connect a daughterboard with a radio module operating in the hundreds of MHz frequency range, but in our project we are going to utilize a NFC daughterboard which works in the 13.56 MHz frequency band. We believed that it would be relatively simple to change to the daughterboard which will be described in section 3.3.2.

3.3.1 Motherboard

Figure 3-1 and Figure 3-2 shows the motherboard that is responsible for providing power and performing the computing required of our embedded platform. The board is designed around a Texas Instruments MSP430F5437A [30] microcontroller (MCU). This processor (shown in Figure 3-2) includes two SPIs: one is used to connect the Ethernet controller and the other connects to a

[*] Serial Peripheral Interface, see section 3.5

daughterboard of the user’s choice. The Ethernet controller is an ENC28J60 [31] chip which communicates with the MCU via the SPI.

The MCU supports two programming interfaces either Bootstrap Loader (BSL) or Joint Test Action Group (JTAG). Our board only included with the JTAG interface and therefore this is the interface we used to install the boot loader code.

With the help of the jumpers (shown in Figure 3-1) a user can choose between getting power from PoE or an external DC power supply. The TL2575HV (shown in Figure 3-2) is a step-down converter [32] which makes it possible to use any DC voltage supply between 3.3V and 60V. In our project we will only utilize PoE (as described in section 2.5). The Ethernet cable from the motherboard needs to be connected to the power sourcing equipment (PSE). This PSE function is generally provided by a switch or router with PoE functionality. The TPS2375 [33] chip takes care of the PoE signaling to tell the host (PSE) the amount of power that it requires. Our board is designed to be a class 1 Powered Device (PD); hence its maximum power is 3.84W. Based upon our initial estimates of the amount of power that we will require, 3.84W is more than enough power.

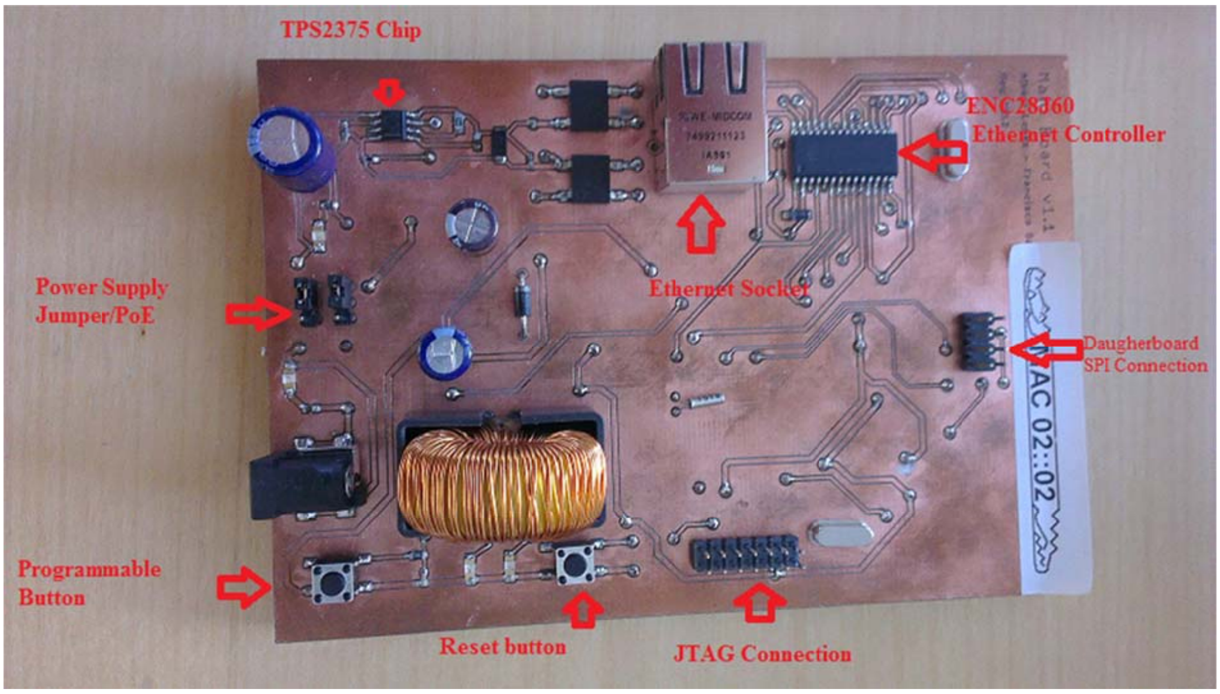


Figure 3-1: **The front side of the motherboard**

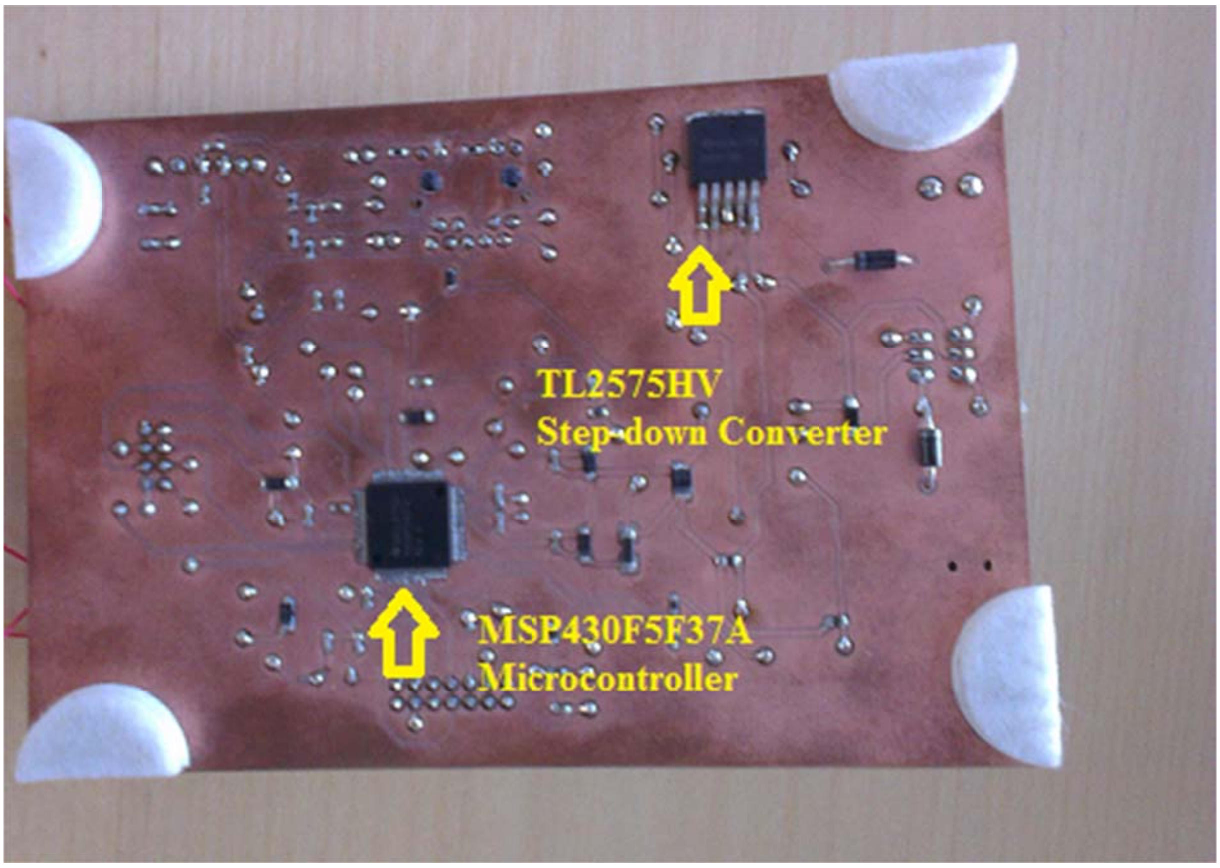
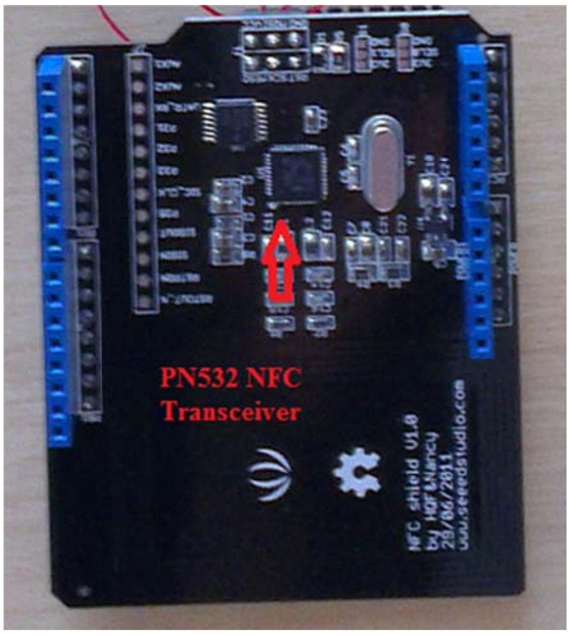


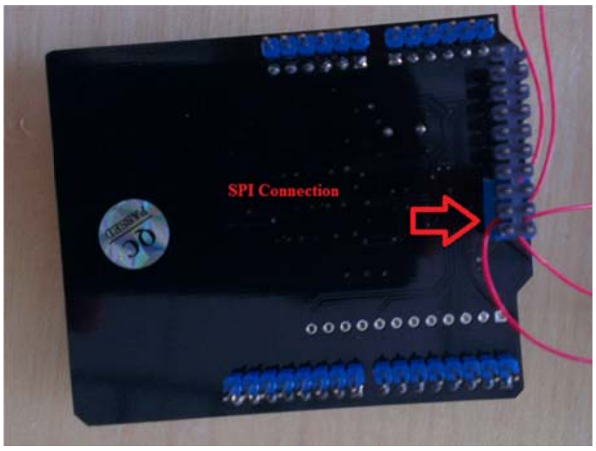
Figure 3-2: **The back side of the motherboard**

3.3.2 Daughterboard (NFC Shield v1.0)

For our daughterboard we selected an NFC shield version 1.0 from *Seeedstudio* [34]. This board (shown in Figure 3-3) is an NFC transceiver that contains all the necessary hardware components for communication with any motherboard that supports SPI.

The shield is built around the popular and widely used NXP PN532 chip. The PN532 provides a complete module for contactless communication in the 13.56 MHz band. The NFC shield also has a built in Printed Circuit Board (PCB) antenna and supports both 3.3V and 5V power supply with the help of a Texas Instruments TXB0104 [35] level translator. This NFC shield was created as a breakout board (i.e., shield) to connect to *Arduino* motherboards. However, because of its low price, it use of a widely used NFC chip, on-board antenna, and the possibility to link it to our motherboard via SPI we selected this board for our project.





(a) **(b)**

Figure 3-3: The front side (a) and the backside (b) of the NFC shield

3.4 Connecting the platform to the network

In this section we explain how we configured the DHCP and the TFTP server. As the system must be connected to a network in order to be able to identify and manage keys, hence the motherboard has to be connected to a network. To provide the motherboard's network interface with an IP address we configured a DHCP server to provide an IP-address to the motherboard based upon a MAC address that we programmed into the network boot loader. As described in section 2.1.1 the actual application to be executed will be transferred at boot time from a TFTP server. Therefore, we also needed to configure a TFTP server on the network. To ensure that the motherboard correctly asks for an IP address and an application to run we used the TFTPboot* program, that Javier Lara Peinado, a previous master student, wrote. We loaded his code into the motherboard using a MSP-FET430UIF JTAG (section 3.2.2) programmer connected to one of our laptop computers.

3.4.1 Configuring the DHCP server

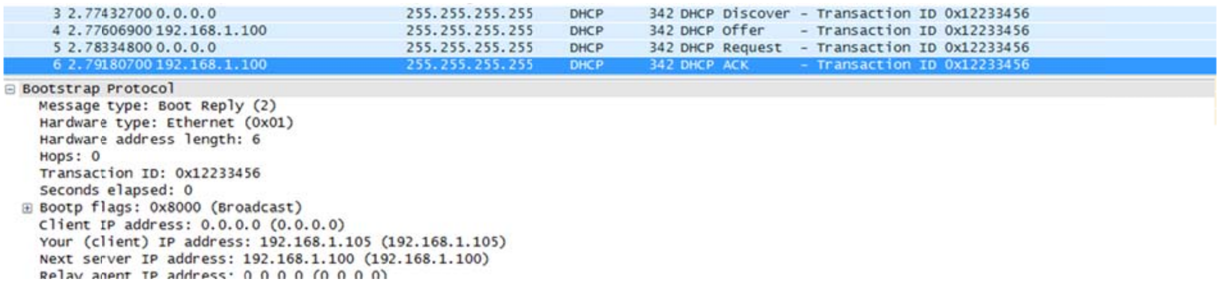
There are many different ways to realize a DHCP server. For example, many routers have a built-in DHCP server. Alternatively, DHCP server software can be installed and configured on a computer that is already connected to the network to have it act as a DHCP server for the network. In order to keep the costs of the product as low as possible we have assumed that the site where the smart door lock system will be installed will already have one or more computers attached to a local area network, hence we chose to utilize one of these computers to act as a DHCP server.

For our project we used a desktop computer running *openSUSE* [29] to run the ISC DHCP Server [36]. The configuration of the DHCP server can be made by editing a configuration file for the DHCP

* https://github.com/cazulu/mind-the-gaps/tree/master/TFTPboot

server (see the configuration file in Appendix A) or by using the YaST [37] which is a configuration and installing tool for the openSUSE [29] system.

To verify that the DHCP server works we used *Wireshark (*See section 3.1.1) to monitor the network traffic; thus we could verify that the expected DHCP discover, offer, request, and acknowledge messages were exchanged and that the motherboard received the IP address that we had configured for it. Figure 3-4 shows a screen capture of the DHCP handshake as captured by Wireshark. Note that the DHCP server was running on a computer with the IP address of 192.168.1.100.



```
  3 2.77432700 0.0.0.0            255.255.255.255   DHCP    342 DHCP Discover - Transaction ID 0x12233456
  4 2.77606900 192.168.1.100      255.255.255.255   DHCP    342 DHCP Offer    - Transaction ID 0x12233456
  5 2.78334800 0.0.0.0            255.255.255.255   DHCP    342 DHCP Request  - Transaction ID 0x12233456
  6 2.79180700 192.168.1.100      255.255.255.255   DHCP    342 DHCP ACK      - Transaction ID 0x12233456
⊟ Bootstrap Protocol
    Message type: Boot Reply (2)
    Hardware type: Ethernet (0x01)
    Hardware address length: 6
    Hops: 0
    Transaction ID: 0x12233456
    Seconds elapsed: 0
  ⊞ Bootp flags: 0x8000 (Broadcast)
    Client IP address: 0.0.0.0 (0.0.0.0)
    Your (client) IP address: 192.168.1.105 (192.168.1.105)
    Next server IP address: 192.168.1.100 (192.168.1.100)
    Relay agent IP address: 0.0.0.0 (0.0.0.0)
```

Figure 3-4: A screen capture of the DHCP process

3.4.2 Configuring the TFTP server

As mentioned before a TFTP server simplifies the loading of a new application into the board. For this purpose we set up a TFTP server. We decided to set up the TFTP server on the same computer that was acting as the DHCP server. The TFTP server was also configured through YaST in the openSUSE PC dekstop. By using *Wireshark* on the workstation that was hosting both the TFTP and DHCP server, we verified that the TFTP server worked by transferring files between two PC`s (one of which was the TFTP server).

When connecting the motherboard to the network, the motherboard should initially make a DHCP request and then download the configured application from the TFTP server. However in our initial experiments this did not happen faultlessly. This is also later explained in section 4.3.

3.5 Serial Peripheral Interface (SPI)

SPI is a data link which can operate in full or half duplex mode. This interface is commonly implemented by microprocessors. Using either a 4-wire (or 3-wire) connection SPI allows a microprocessor (microcontroller) to communicate with other devices such as sensors or other microprocessors that support SPI.

The SPI protocol is designed to support a single master device to communicate with one or more slaves connected to a serial bus. The master (the microprocessor) initiates communication and provides a clock for synchronization. The slave is a device (in our case either a NFC-reader or the ENC28J60 Ethernet controller) which responds to the mater's communication, receives commands and sends data, under the control of the microprocessor. Table 1 shows the signals used by SPI. We will use these names for these signals in our description[*]. SPI utilizes 8 bit bytes [38].

Table 1: SPI signals

Signal Name	Description
SCLK	Serial Clock
MOSI/SIMO	Master Output Slave Input
MISO/SOMI	Master Input Slave Output
SS (CE)	Slave Select (Chip Enable)

[*] Note that there are other commonly used names for these signals.

As mentioned above, the master initializes the communication between the devices. Before initiating this communication the master first configures the serial clock of the selected SPI to a frequency which is less or equal to the maximum frequency supported by the slave device. For example, in our case this frequency is either 5 MHz for the NFC-reader or 8 MHz for the ENC28J60 Ethernet controller. The code[*] to configure the SPI interface used to connect to the Ethernet controller is shown below:

[*] Extracted from the config.c file of the TFTPboot program. Note that the code has been reformatted for inclusion in this thesis.

From hardware_board.c

```
// ENC28J60
#define ETH_CS                   BIT0
#define ETH_CS_IN                P3IN
#define ETH_CS_OUT               P3OUT
#define ETH_CS_DIR               P3DIR
#define ETH_CS_REN               P3REN

#define ETH_INT                  BIT2
#define ETH_INT_IN               P1IN
#define ETH_INT_DIR              P1DIR
#define ETH_INT_OUT              P1OUT
#define ETH_INT_REN              P1REN
#define ETH_INT_IES              P1IES
#define ETH_INT_IE               P1IE
#define ETH_INT_IFG              P1IFG

#define ETH_RST                  BIT3
#define ETH_RST_OUT              P1OUT
#define ETH_RST_DIR              P1DIR
```

From config.c:

```
// ENC28J60 SPI port
#define ETH_SIMO                 BIT1
#define ETH_SOMI                 BIT2
#define ETH_SCLK                 BIT3
#define ETH_SPI_IN               P3IN
#define ETH_SPI_OUT              P3OUT
#define ETH_SPI_DIR              P3DIR
#define ETH_SPI_REN              P3REN
#define ETH_SPI_SEL              P3SEL

void InitializeEthSpi(void)
{
           // Activate reset state
           UCB0CTL1 |= UCSWRST;

           // Configure ports
           ETH_SPI_SEL |= ETH_SCLK + ETH_SIMO + ETH_SOMI;
          // Special functions for SPI pins

           ETH_SPI_DIR |= ETH_SIMO + ETH_SCLK;         // Outputs
           ETH_CS_DIR |= ETH_CS;
           ETH_CS_OUT |= ETH_CS;

           // Configure SPI registers
           UCB0CTL0 |= UCCKPH + UCMSB + UCMST + UCSYNC;
          // Clock phase 0, Clock pol 0, 8-bit
           // MSB first, Master mode, 3-pin SPI, Synch
           UCB0CTL1 |= UCSSEL_2; // SMCLK clock source
           UCB0BR0 = 0;          // No Prescaler (8MHz)
           UCB0BR1 = 0;
           UCA0MCTL = 0;

           // Deactivate reset state
           UCB0CTL1 &= ~UCSWRST;
}
```

The master then selects the desired slave by pulling the SS line to the “low” state. The slaves that have not been activated by the master using its slave select will disregard the serial clock and MOSI signals from the master. In this manner the master selects only one slave at the time [39].

When the slave wishes to communicate with the microprocessor (master) the slave can use an interrupt line to indicate that an event has occurred. Otherwise the master needs to poll the slave(s) to see if it (they) have any input.

The SPI interface can work in different clock modes depending on the microcontroller you want to connect, there are four clock modes. The clock mode depends on the configuration of the clock polarity and phase; Figure 3-5 shows a timing diagram of the different clock modes.

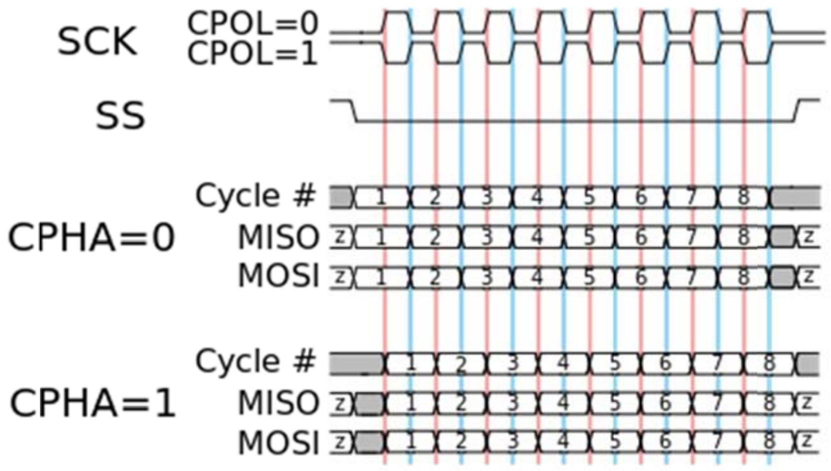


Figure 3-5: **SPI timing diagram, the vertical red line represent CPHA = 0 and the blue line CPHA = 1**

At *CPOL = 0* (clock polarity) the base value of the clock is zero and at *CPOL = 1* the base value of the clock is one. When *CPOL = 0* is chosen at *CPHA = 0* (clock phase) data are captured on the clock's rising edge and data are transmitted on the falling edge, with *CPHA = 1* data are captured on the clock's falling edge and data are transmitted on the clock's rising edge. When *CPOL = 1* and *CPHA = 0* data are captured on clock's falling edge and data is transmitted on the rising edge, while with *CPHA = 1* data are captured on the clock's rising edge and data is transmitted on the falling edge of the clock.

These settings result in a total of four clock modes that can be configured. Table 2 summarizes these different modes.

Table 2: **The four different clock modes**

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

4 Implementation and Analysis

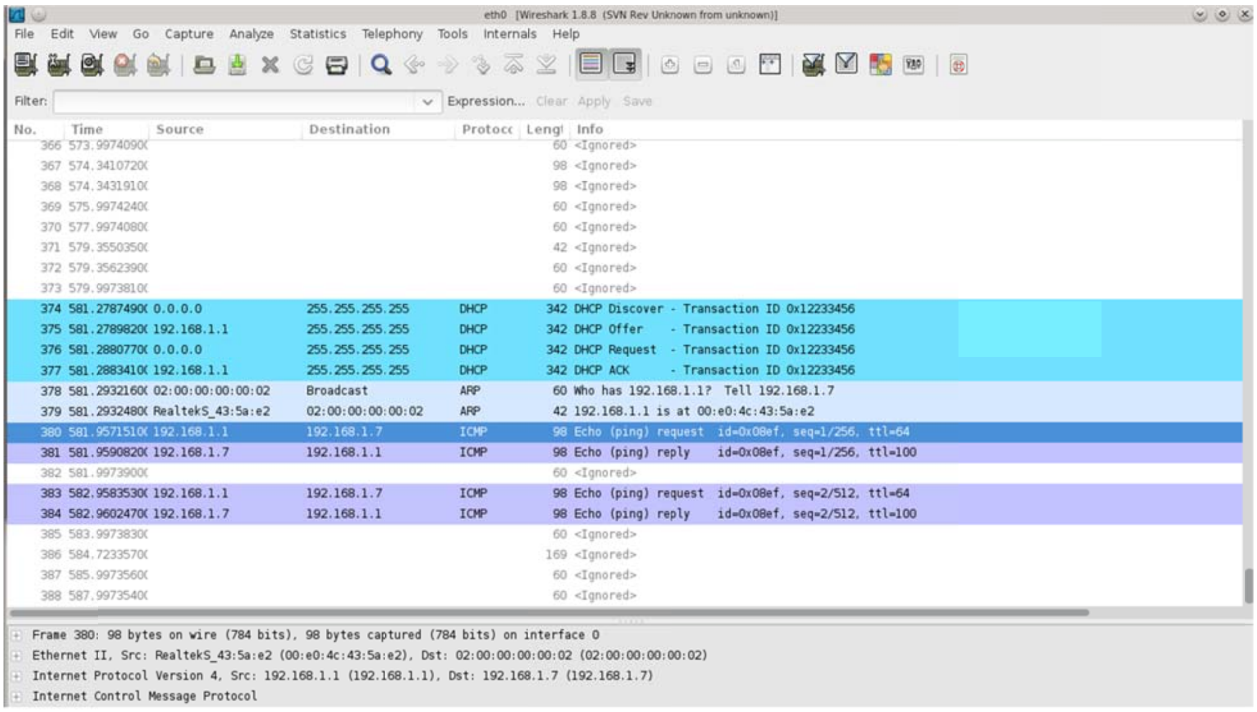
In this chapter we will explain the steps we toke to achieve the goals we set in chapter 3, which included programming, reading and a lot of debugging. All the files used in this project can be found at the Dropbox link https://www.dropbox.com/sh/43h519564hspkw2/RcXkXGtCqQ . The appendix gives a short description of what the folders contain and what they were used to do.

This chapter begins with a description in section 4.1 of how we could ping the motherboard to verify network connectivity and in section 4.2 we describe how we transferred and received UDP packets. In section 4.3 we describe the idea of network booting. The SPI configurations are described in section 4.4 and the debugging of the connection between the board and shield are described in section 4.6. The NFC tags, the android application and the python server are described in section 4.5. The whole system working together with a Python UDP server is described in 4.8. The chapter concludes with a description of the complete smart door lock system.

4.1 Verifying the network connection

As we use UDP packets to send data to and from the motherboard of our system we wanted to ensure that our system was communicating correctly with our DHCP server. In order to do this we used Wireshark to view the traffic between the DHCP server and our board. After establishing that the board had requested and received an IP address we used the program ping on the PC to send an ICMP echo request message to our board just to see if it would respond correctly. This verified that the network interface of the board had no problems receiving and sending ICMP packets and that the network stack was running on the microprocessor.

Figure 4-1 displays an ICMP Echo request from the computer (that was also acting as our DHCP Server) with the IP address of 192.168.1.1 and a response from our board with the IP address of 192.168.1.7.


eth0 [Wireshark 1.8.8 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time        Source              Destination        Protocc Lengt Info
 366 573.9974090(                                                60 <Ignored>
 367 574.3410720(                                                98 <Ignored>
 368 574.3431910(                                                98 <Ignored>
 369 575.9974240(                                                60 <Ignored>
 370 577.9974080(                                                60 <Ignored>
 371 579.3550350(                                                42 <Ignored>
 372 579.3562390(                                                60 <Ignored>
 373 579.9973810(                                                60 <Ignored>
 374 581.2787490( 0.0.0.0            255.255.255.255    DHCP     342 DHCP Discover - Transaction ID 0x12233456
 375 581.2789820( 192.168.1.1        255.255.255.255    DHCP     342 DHCP Offer    - Transaction ID 0x12233456
 376 581.2880770( 0.0.0.0            255.255.255.255    DHCP     342 DHCP Request  - Transaction ID 0x12233456
 377 581.2883410( 192.168.1.1        255.255.255.255    DHCP     342 DHCP ACK      - Transaction ID 0x12233456
 378 581.2932160( 02:00:00:00:00:02  Broadcast          ARP       60 Who has 192.168.1.1?  Tell 192.168.1.7
 379 581.2932480( RealtekS_43:5a:e2  02:00:00:00:00:02  ARP       42 192.168.1.1 is at 00:e0:4c:43:5a:e2
 380 581.9571510( 192.168.1.1        192.168.1.7        ICMP      98 Echo (ping) request  id=0x08ef, seq=1/256, ttl=64
 381 581.9590820( 192.168.1.7        192.168.1.1        ICMP      98 Echo (ping) reply    id=0x08ef, seq=1/256, ttl=100
 382 581.9973900(                                                60 <Ignored>
 383 582.9583530( 192.168.1.1        192.168.1.7        ICMP      98 Echo (ping) request  id=0x08ef, seq=2/512, ttl=64
 384 582.9602470( 192.168.1.7        192.168.1.1        ICMP      98 Echo (ping) reply    id=0x08ef, seq=2/512, ttl=100
 385 583.9973830(                                                60 <Ignored>
 386 584.7233570(                                               169 <Ignored>
 387 585.9973560(                                                60 <Ignored>
 388 587.9973540(                                                60 <Ignored>

Frame 380: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: RealtekS_43:5a:e2 (00:e0:4c:43:5a:e2), Dst: 02:00:00:00:00:02 (02:00:00:00:00:02)
Internet Protocol Version 4, Src: 192.168.1.1 (192.168.1.1), Dst: 192.168.1.7 (192.168.1.7)
Internet Control Message Protocol
```

Figure 4-1: Wireshark capture of the ICMP request and the respond back.

4.2 UDP Echo software

In order to ensure that we could send and receive UDP packets containing our own data, we decided to start by implemented a simple program to echo a string that we sent to the board from the PC. Since our board used the ENC28J60 for its network communication we had to learn the steps that are necessary for sending UDP Packets, so by looking into frequency scanner project file in the master project of Javier Lara Peinado [4] and also reading into the UDP.c file provided by Microchip we finally managed to understand the steps required for our goal. These steps are described below.

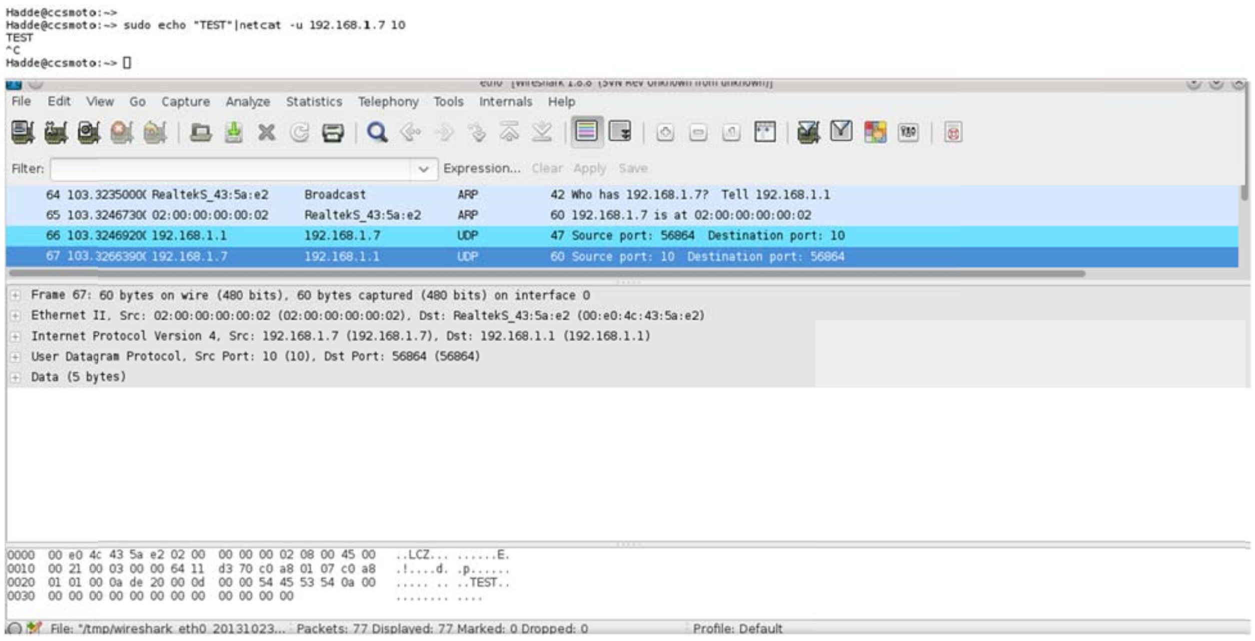
In order to send a UDP packet we must:

1. Call the "ARPResolve" routine to obtain the MAC of the IP destination we wish to send the UDP datagram to.
2. Call UDPOpen to open a socket to be used to send our UDP Packets.
3. Call UDPisPutReady (sets the current socket as the active socket, and determines how many bytes can be written to this UDP socket). This step is important because the ENC28J60 provides the buffering for the IP packet (or packets), hence the processor does not have to buffer the complete datagram – thus reducing the amount of RAM memory that the program needs.
4. Call UDPPut (or UDPPutArray) for building and storing the data into the packet within the ENC28J60.
5. Call UDPFlush to send our packet to the desired destination.
6. Call UDPClose to close the socket (this is not mandatory)

When the Ethernet controller receives UDP packets, then we need to use the following two functions:

1. Call UDPisGetReady (sets the current socket as the active one, and determines how many bytes can be **read** from the UDP socket) and
2. Call UDPGet (reads a byte from the currently active socket).

For this testing we used a shell window on the PC that was hosting the DHCP server. We invoked our test program with the destination IP address and the port of the board to send our text message and received the expected response back from the board. The result of test is shown in Figure 4-2, where we sent the string "TEST" and received this response echoed back from our board. We could have chosen to use the Transmission control protocol (TCP) for our data packets, but since we also wanted the TFTP in our project and it already used the UDP for its file transfers, we thought it would be better to stay with the UDP protocol, also the UDP protocol has faster file transfer speed and if case of a packet drop then one would have to just resend the packet.



```
Hadde@ccsmoto:~>
Hadde@ccsmoto:~> sudo echo "TEST"|netcat -u 192.168.1.7 10
TEST
^C
Hadde@ccsmoto:~> []
```

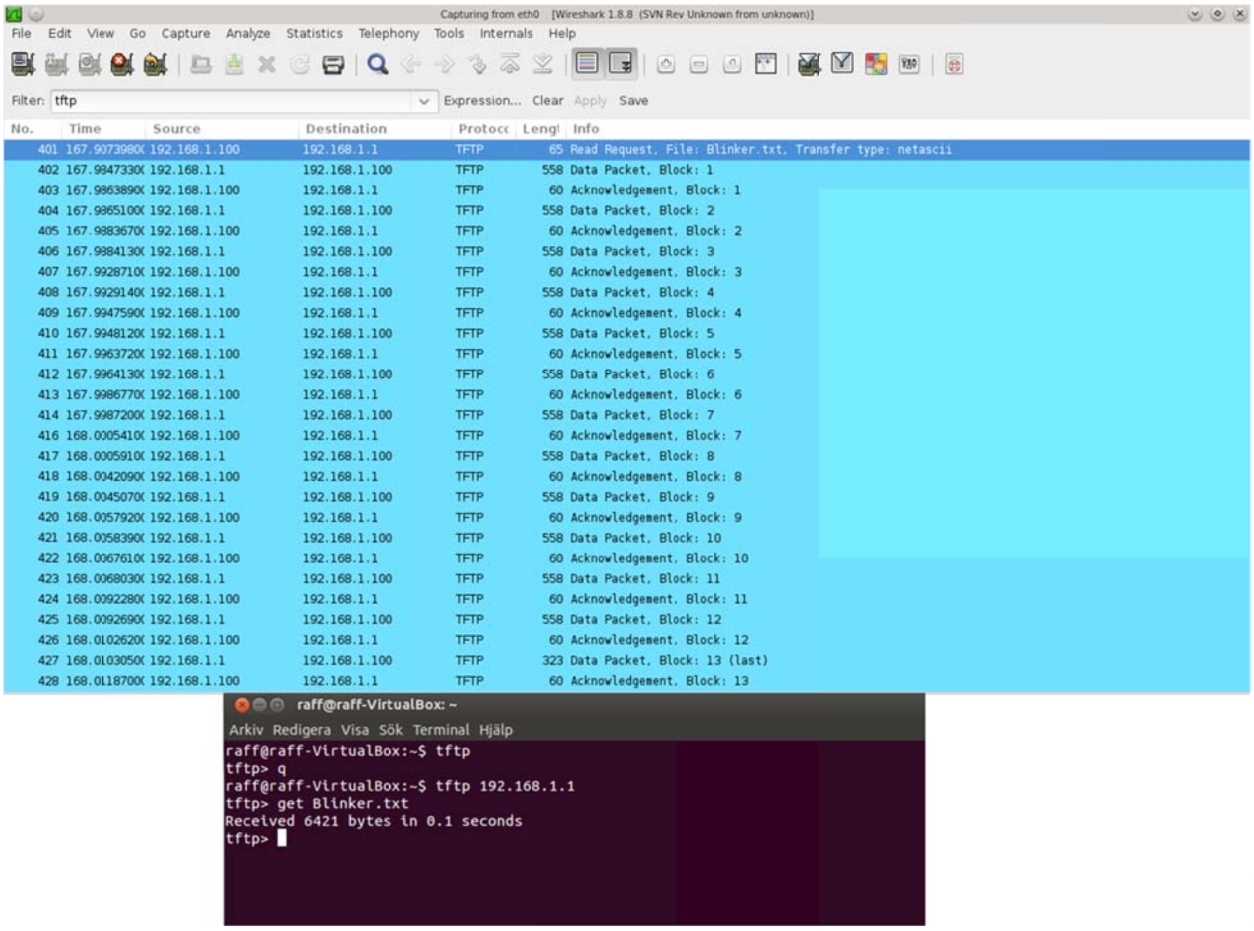
```
64 103.3235000( RealtekS_43:5a:e2   Broadcast          ARP   42 Who has 192.168.1.7?  Tell 192.168.1.1
65 103.3246730( 02:00:00:00:00:02   RealtekS_43:5a:e2  ARP   60 192.168.1.7 is at 02:00:00:00:00:02
66 103.3246920( 192.168.1.1         192.168.1.7        UDP   47 Source port: 56864  Destination port: 10
67 103.3266390( 192.168.1.7         192.168.1.1        UDP   60 Source port: 10  Destination port: 56864

Frame 67: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: 02:00:00:00:00:02 (02:00:00:00:00:02), Dst: RealtekS_43:5a:e2 (00:e0:4c:43:5a:e2)
Internet Protocol Version 4, Src: 192.168.1.7 (192.168.1.7), Dst: 192.168.1.1 (192.168.1.1)
User Datagram Protocol, Src Port: 10 (10), Dst Port: 56864 (56864)
Data (5 bytes)

0000  00 e0 4c 43 5a e2 02 00  00 00 00 02 08 00 45 00   ..LCZ... ......E.
0010  00 21 00 03 00 00 64 11  d3 70 c0 a8 01 07 c0 a8   .!....d. .p......
0020  01 01 00 0a de 20 00 0d  00 00 54 45 53 54 0a 00   ..... .. ..TEST..
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....

File: "/tmp/wireshark_eth0_20131023...  Packets: 77 Displayed: 77 Marked: 0 Dropped: 0     Profile: Default
```

Figure 4-2: The dark blue line shows the message TEST being echoed back from our board.

4.3 Network Booting

As mentioned in section 2.3 we knew the TCP/IP stack implemented in our board supported the TFTP protocol. We wanted to take advantage of to automatically transfer our code to the board immediately after the board has received an IP address from a DHCP server. The thought of this was that the board would directly load its code through the network, thus avoiding the need to connect a programmer to the board. In the case of deployed system this would mean that each device could get the latest version of the software when it next rebooted or when commanded to.

We set up a TFTP server which included an example file that was meant to be sent to our board immediately after it received an IP address. Before we tried to send this file to our board we first tried to send the file to another PC connected to the same TFTP server. The other PC was running *Ubuntu 12.04 LTS* in a Virtual-box [40]. Figure 4-3 displays the Wireshark output as Ubuntu requests and receives the file from our TFTP server.


```
No.  Time          Source          Destination     Protocc Lengt Info
401 167.9073980( 192.168.1.100   192.168.1.1     TFTP     65 Read Request, File: Blinker.txt, Transfer type: netascii
402 167.9847330( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 1
403 167.9863890( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 1
404 167.9865100( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 2
405 167.9883670( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 2
406 167.9884130( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 3
407 167.9928710( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 3
408 167.9929140( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 4
409 167.9947590( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 4
410 167.9948120( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 5
411 167.9963720( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 5
412 167.9964130( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 6
413 167.9986770( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 6
414 167.9987200( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 7
416 168.0005410( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 7
417 168.0005910( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 8
418 168.0042090( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 8
419 168.0045070( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 9
420 168.0057920( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 9
421 168.0058390( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 10
422 168.0067610( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 10
423 168.0068030( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 11
424 168.0092280( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 11
425 168.0092690( 192.168.1.1     192.168.1.100   TFTP    558 Data Packet, Block: 12
426 168.0102620( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 12
427 168.0103050( 192.168.1.1     192.168.1.100   TFTP    323 Data Packet, Block: 13 (last)
428 168.0118700( 192.168.1.100   192.168.1.1     TFTP     60 Acknowledgement, Block: 13
```

```
raff@raff-VirtualBox:~$ tftp
tftp> q
raff@raff-VirtualBox:~$ tftp 192.168.1.1
tftp> get Blinker.txt
Received 6421 bytes in 0.1 seconds
tftp> 
```

Capturing from eth0 [Wireshark 1.8.8 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: tftp
Expression... Clear Apply Save
raff@raff-VirtualBox: ~
Arkiv Redigera Visa Sök Terminal Hjälp

Figure 4-3: File Blinker.txt is requested from 192.168.1.100 and successfully received.

Now that we knew that the TFTP and DHCP configuration have had no problems, our next step was to have the system (motherboard) request the file from the TFTP server that it learned in the DHCP response from the DHCP server. However, here we ran into a problem as the board only requested a small part of the file and stored it in the first block of the memory. After this, the transfer kept looping requesting the first block because the motherboard never acknowledges the first block, which made the TFTP server resend the first block. Because of this our file was not successfully transfer to our board.

Due to other priorities in our project we never succeeded in finding a solution to this problem. For this reason debugging the TFTP loading of an application remains as future work.

4.4 SPI Drivers configuration

The declarations that define the pins to be used for the interface the NFC daughter card are shown below. These ports use the UCA1 register on the MSP430.

```
// RF SPI port
#define RF SIMO                     BIT6
#define RF SOMI                     BIT7
#define RF SPI IN                   P5IN
#define RF SPI OUT                  P5OUT
#define RF SPI DIR                  P5DIR
#define RF SPI REN                  P5REN
#define RF SPI SEL                  P5SEL

#define RF SCLK                     BIT6
#define RF SPI SCLK DIR             P3DIR
#define RF SPI SCLK SEL             P3SEL
```

To configure the SPI interface used to connect the daughter card (NFC-shield) to the master (MSP430-microcontroller) there is some settings that need to be set for the SPI interface to work properly. As mentioned earlier the daughter card is based on a PN532 microchip which needs to have the settings [41] shown in Table 3.

Table 3: PN532 SPI settings

The mode used for the clock is **Mode 0**
Data is always sampled on the **first clock edge of SCK**
SCK is active **High**
The data order used is **LSB** first

These settings are set in the control register of the SPI interface we are using, in our case UCA1CTL0. We also have to choose a clock source for the signals to be sent. The unified clock system (**UCS**) module in the MSP430 provides various clocks. There is up to five clock sources to choose from [30]:

XT1CLK	Low-frequency or high-frequency oscillator that can be used with low frequency 32768 Hz watch crystals, standard crystals, resonators, or external clock sources in the 4 MHz to 32 MHz range.
VLOCLK	Internal very low power, low frequency oscillator with a typical frequency of 10 kHz.
REFOCLK	Internal, trimmed, low-frequency oscillator with 32768 Hz typical frequency.
DCOCLK	Internal digitally-controlled oscillator (*DCO*)
XT2CLK	Optional high-frequency oscillator that can be used with standard crystals, resonators or external clock sources in the 4 MHz to 32 MHz range.

Three clock signals are available from the UCS module:

ACLK	Auxiliary clock (32 kHz).
MCLK	Master clock (8 MHz).
SMCLK	Subsystem master clock (8MHz).

These three clocks (ACLK, MCLK, SMCLK) are software selectable as XT1CLK, REFOCLK, VLOCKL, DCOCLK, and when available, XT2CLK. ACLK and SMCLK are software selectable by individual peripheral modules and are available externally at a pin, and MCLK is used by the CPU and system. All these clocks can be divided by 1, 2, 4, 8, 16, or 32 to provide the desired clock frequency. As mentioned in section 3.4 the maximum SPI clock frequency for the PN532 is 5 MHz, to generate the desired clock frequency we chose to use the SMCLK as our clock source and divided it by 2 to get a 4 MHz clock source which the PN532 can utilize.

4.5 Implementing the SPI functions

SeeedStudio, the developer of the NFC-shield, provided some example source code for their NFC-Shield for the Arduino board. This source code does not include the actual transmit and receive functions for the SPI interface as these examples were written for the Arduino board which provides SPI functions via a SPI library. Because we are using the MSP430-microcontroller we had to implement these SPI functions for the MSP430 in order to make the example code work.

Energia [42] is a prototyping platform with the goal to bring the Arduino framework to the Texas Instruments MSP430. However, Energia only supports the MSP430 LaunchPad and FraunchPad, which are not based on the MSP430f5437a (the microcontroller that we are using). The main difference is the pin map of the boards and the registers, as mentioned in section 4.4 are we using the UCA1CTL0 register for the SPI connection (NFC-shield), as the LaunchPad and FraunchPad may use other interfaces. However, the transmit and receive functions should work with a change in the register.

Looking at the example source code from SeeedStudio we could determine which SPI function was critical for the example code to work on the MSP430. The most critical SPI function that we needed to implement was the transmit function[*] (which is basically the same as the receive function) and the slave select (SS) function to select the SPI slave to communicate with.

The transmit and receive functions are the same function, as when we want to receive data we simply transmit a zero and then read the response. When we transmit data we send data and ignore the response.

We had to write the SS function ourselves because the one provided from Energia did not suit the microcontroller we are using. This function simply changes the SS pin from HIGH to LOW when the slave is to be selected, and the reversed when unselected.

4.6 Connecting the daughterboard and debugging

We used SPI to connect the daughterboard (NFC Shield) to the motherboard. In order to do this we had to connect the connections mentioned in Table 3-1, as well as provide power and ground. The motherboard brought the SPI signals, 3.3 V power, and ground to an easy to access connector. Similarly the NFC shield's SPI pins, 3.3 V power input, and ground were brought to a convenient to access connector. This would seem to have made it easy to connect the two boards together. Figure 4-4 shows these connections between these two boards.

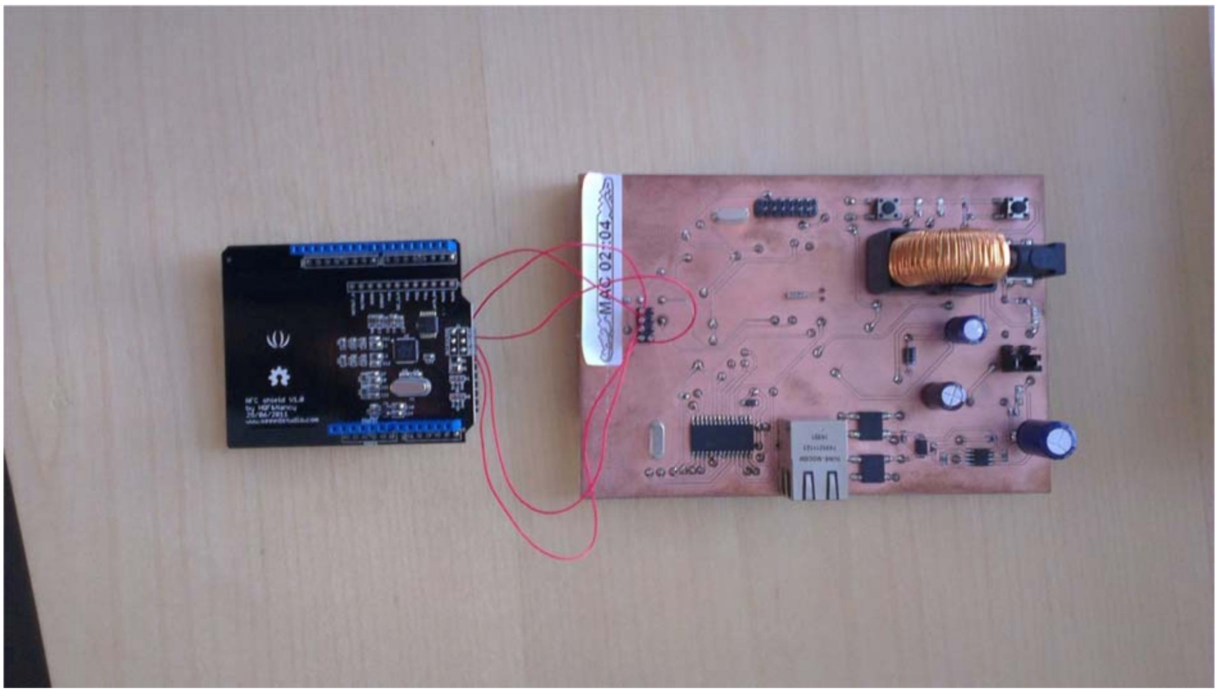


Figure 4-4: The SPI, power, and ground connection between the boards

The NFC shield (daughterboard) had a built in program that gets command from a SPI master device and sends back a respond to that command. For example; if the master device (MSP430) sends the *get-firmware-version* command to the NFC shield, the shield should respond with the firmware version it is running. This method was used to verify that the SPI connection was working. We used the oscilloscope, described in section 3.2.1, to see if there was any response from the NFC shield, Figure 4-5 shows the command being sent from the MSP430 (line 2) and no response from the NFC shield (line 3). Note that line 1 on the display shows the clock signal from the SPI master.

[*] See appendix

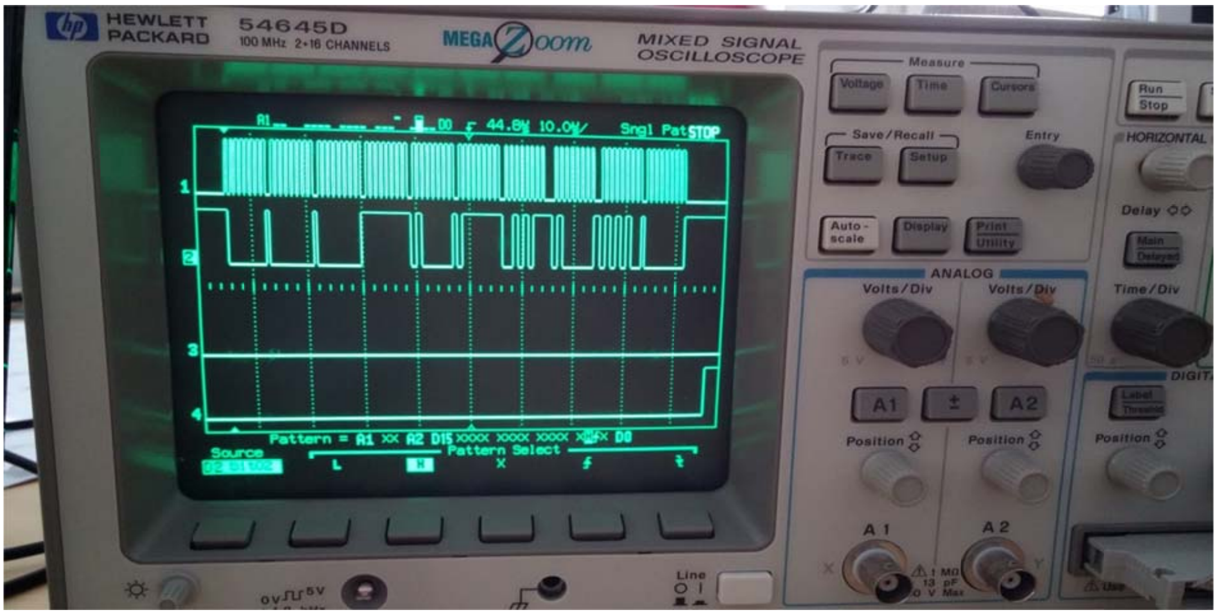


Figure 4-5: **Get Firmware Version command on the motherboard without any response**

As the NFC shield was designed to be used with the Arduino board, there was a suspicion that there was another connection in addition to the SPI signals, 3.3 V power, and ground that needed to be connected. To find that connection we decided to attach the NFC shield to an Arduino board and then looked at the SPI signals and other signals on the oscilloscope. However, there were approximately 58 connections (8 of which are grounds) between the shield and the Arduino board. As this seemed to be too many for us to examine to find which connection was missing, we looked at the board layout of the NFC shield to see exactly what each of the connectors was connected to. Fortunately, the developers of the NFC shield (*SeeedStudio*) provided a board layout of the board in Eagle[*] format (shown in Figure 4-6) and they provided a schematic for the board (also in Eagle format).

[*] Easily Applicable Graphical Layout Editor (EAGLE) is an electronic design automation application to make schematics of PCBs.

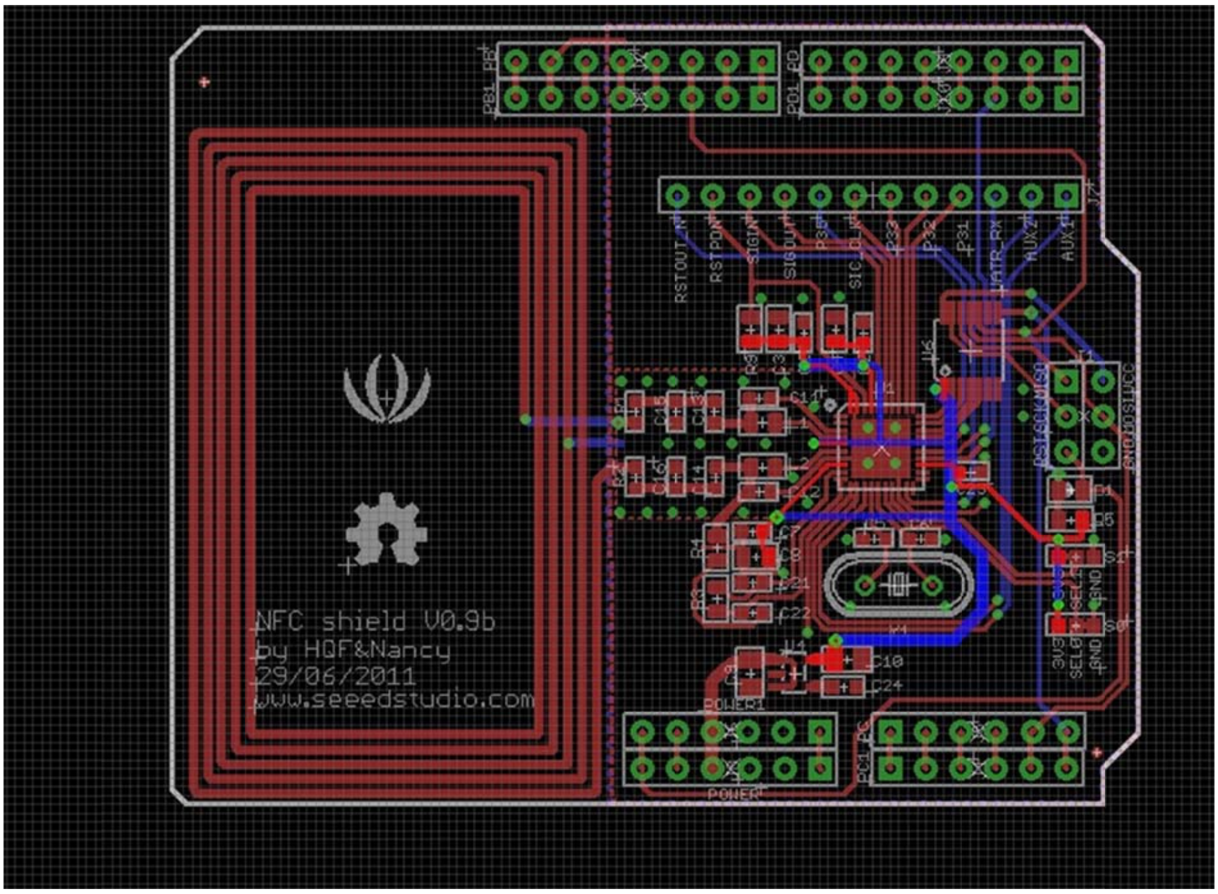


Figure 4-6: Board layout of the NFC shield (displayed by Eagle)

In addition, to the SPI connections on the right end of the board (to which we connected when interfacing to our motherboard) when the shield is plugged into to the Ardunio board the connectors across the top and the bottom of the board are connected. Based upon the schematic we believed that many of these connectors were connected to different parts of the logic on this shield. However, after examining the board layout we could exclude most of the connectors on the board because they did **not** actually connect to anything on the shield (despite what the schematics suggested). Moreover, after examining which signals were interfaced to at the SPI, power, and ground connector on the right side of the board one of the connections that we did **not** have was the connection from the power connector on the bottom of the board. Following this connection lead us to the TXB0104PWR voltage-level translator [43] (marketed U6 on the schematic) which gets power from two sources, the SPI interface, power, and ground connector and from the Arduino power interface. Looking at the datasheet for the voltage-level translator we learned that the voltage-level translator needs two power sources to function. However, with only a single power source from the SPI interface, power, and ground connector this effectively isolated all of the SPI signals from the PN532 chip!

After connecting power to the level translator (using the dark green wire shown in Figure 4-7) we sent the *get firmware version* command[*] to the NFC shield from the Arduino board to see if there was any response. Figure 4-8 shows the command being sent on line 2 and that there a response on line 3.

[*] The definition of the commands for the PN532 NFC shield can be found in the file gw.h in the project folder

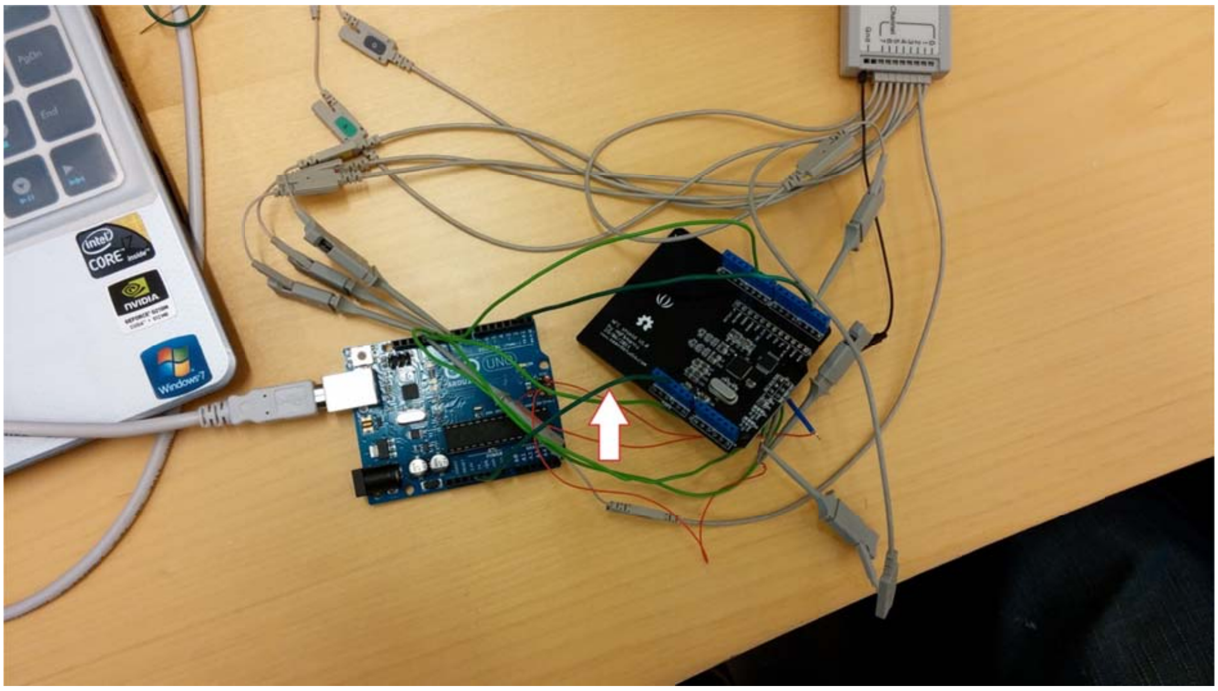


Figure 4-7: **The connection between the Arduino board and the NFC shield with the oscilloscope connected to examine the signals**

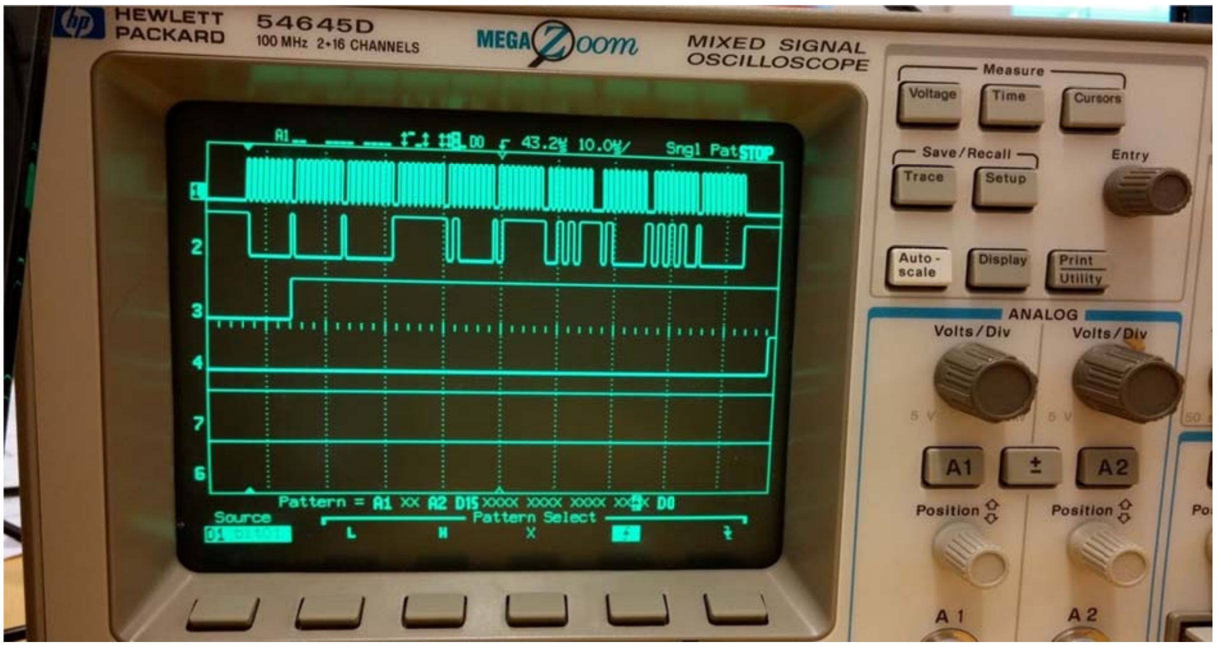


Figure 4-8: **Get Firmware Version command on the Arduino board with response**

Now that all the connection had been found and tested with the Arduino board we connected the NFC shield to the MSP430 motherboard and connected the oscilloscope to examine the signals.

Figure 4-9 shows the final, working, communication between the MSP430 and the NFC shield, the green cable on the top of the NFC shield is the additional power connection for the level translator (Note that this is 3.3V.). The signals that are transmitted and received are shown on the screen of the oscilloscope in Figure 4-10.

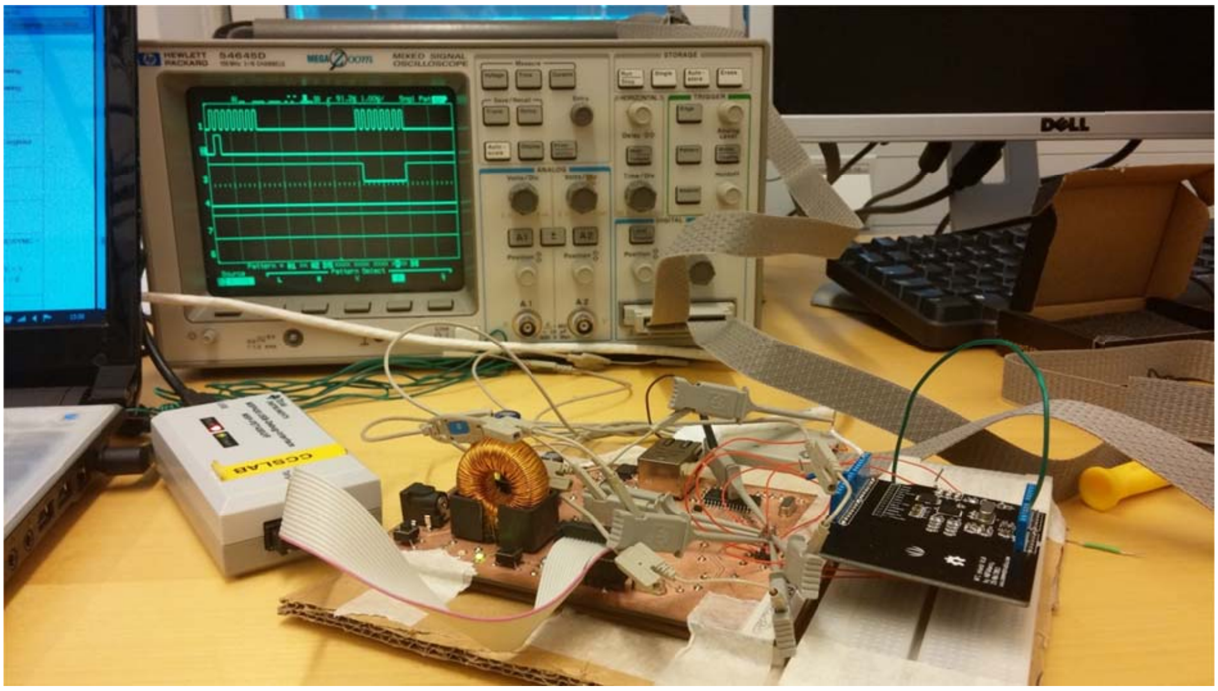


Figure 4-9: The final connection with the oscilloscope connected to examine the signals

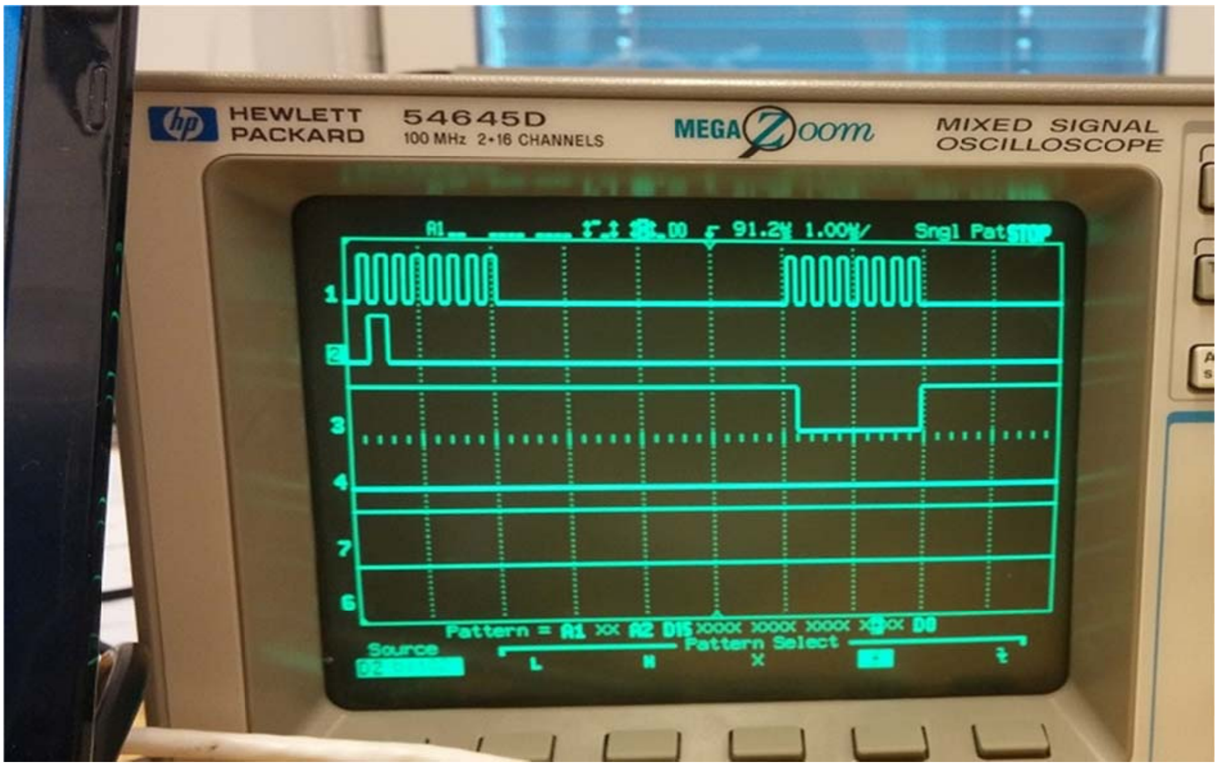


Figure 4-10: Read SPI Status sent from the MSP430 to the NFC shield with a response

Line 1 on the oscilloscope in Figure 4-10 is the clock of the MSP430, line 2 is the master-out-slave-in (MOSI) connection, line 3 is the slave-out-master-in (MISO) connection, line 4 is the slave select (SS) line. Line 2 shows the read SPI status command being sent to the NFC shield, the read SPI command consist of the hexadecimal value of 2, which can be seen on line 2 in Figure 4-10, note that the SPI operates in LSB mode[*]. The command is sent during the first clock sequence and during the second clock sequence the NFC shield replies with its status, in this case the value 1 which means that the SPI device is ready.

4.7 Setting up appropriate triggering of the mixed oscilloscope

An important element of the debugging describe in the previous section was setting up an appropriate trigger – so that one could see the SPI commands and responses.

The signals that are shown on the oscilloscope are in real time, this means that the signals are shown as they are sent and this happens very fast, which makes it hard to inspect the signals. One can pause the oscilloscope on a signal but this method is not very precise or effective as the screen is paused at the moment the button is pressed, which means that there might not be any or the right signals shown on the screen of the oscilloscope. This is why we had to use the trigger function of the oscilloscope.

To set up the oscilloscope to use the trigger function we have to set up a pattern to trigger on, to do this we simply pressed the Pattern button on the oscilloscope, this brings up a new menu to set up the pattern to follow on each line connected to the oscilloscope. We can choose from triggering on a falling edge of a signal or on a rising edge. The oscilloscope that we used only supports triggering on one line, which means that we can only select one signal to follow and edge trigger on, however, the pattern can be configured to include other lines but this only checks if the other lines is in High or Low mode depending on the pattern we set. For example: if we would like to trigger on the first signal when we send data we set up the pattern to trigger on the rising edge of the data line, we also know that the SS line should be in Low state as the slave board is selected, this pattern should freeze the screen on the oscilloscope at the moment when there is a rising edge of the data line **and** the SS line is in Low state. There is a lot of options to set up a pattern, in our case we used the “and**”** operator to choose a pattern, however, one can use the “or”, “exclusive or”, etc. to setup a pattern between the different inputs.

4.8 Android application and NFC Tags

As mentioned in section 2.2 NFC is a contactless communication technology, it allows a data transfer between two NFC enabled devices. An NFC tag could be realized as a sticker which contains a small microchip that can store a small amount of data and transfer it to a NFC enabled device. How much data you can store depends on which type of tag you are using, since different types of tags have different memory capacities. The tags used in our project are MIFARE Classic tags [44] with a memory size of 1 Kilobyte (KB). The MIFARE Classic tags are widely used today in public transportation, electronic toll collection, and in loyalty cards. Our tags are also NFC Data Exchange Format (NDEF) enable and formatable. This means that one can store messages or homepages and even different application-defined actions on the tag to be viewed or used from any NFC enabled device. Since the tags are formatable an application can delete the data stored on the tag and write new content to be stored by the tag. The tags we used also contain a Unique Identifier Number (UID) which we used in our project to identify a specific.

In order to read the UID of each NFC tag we used the Android application TagWriter [45] from the Google play application market. This application was developed by the company NXP Semiconductors [46], the same company as makes the PN532 NFC chip used on the shield.

We implemented an application in the motherboard. This application simply sends a UID read requests to the NFC shield. If an NFC tag is located near the antenna of the NFC shield the NFC shield will get a response from the tag. The application places the tag’s UID into a UDP packet and sends it to our UDP server (a program running on the PC acts as a UDP server). The UDP server compares this tag against entries in a list of acceptable NFC tag UIDs associated with this smart door lock system. If

[*] See section 4.4

the UID of this tag is in the list, then the UDP server sends an UDP packet to the board informing it to either lock or unlock the lock. For our testing we indicated whether access was granted or denied by turning on a green (granted) or orange (denied) LED on the board.

4.9 UDP Server

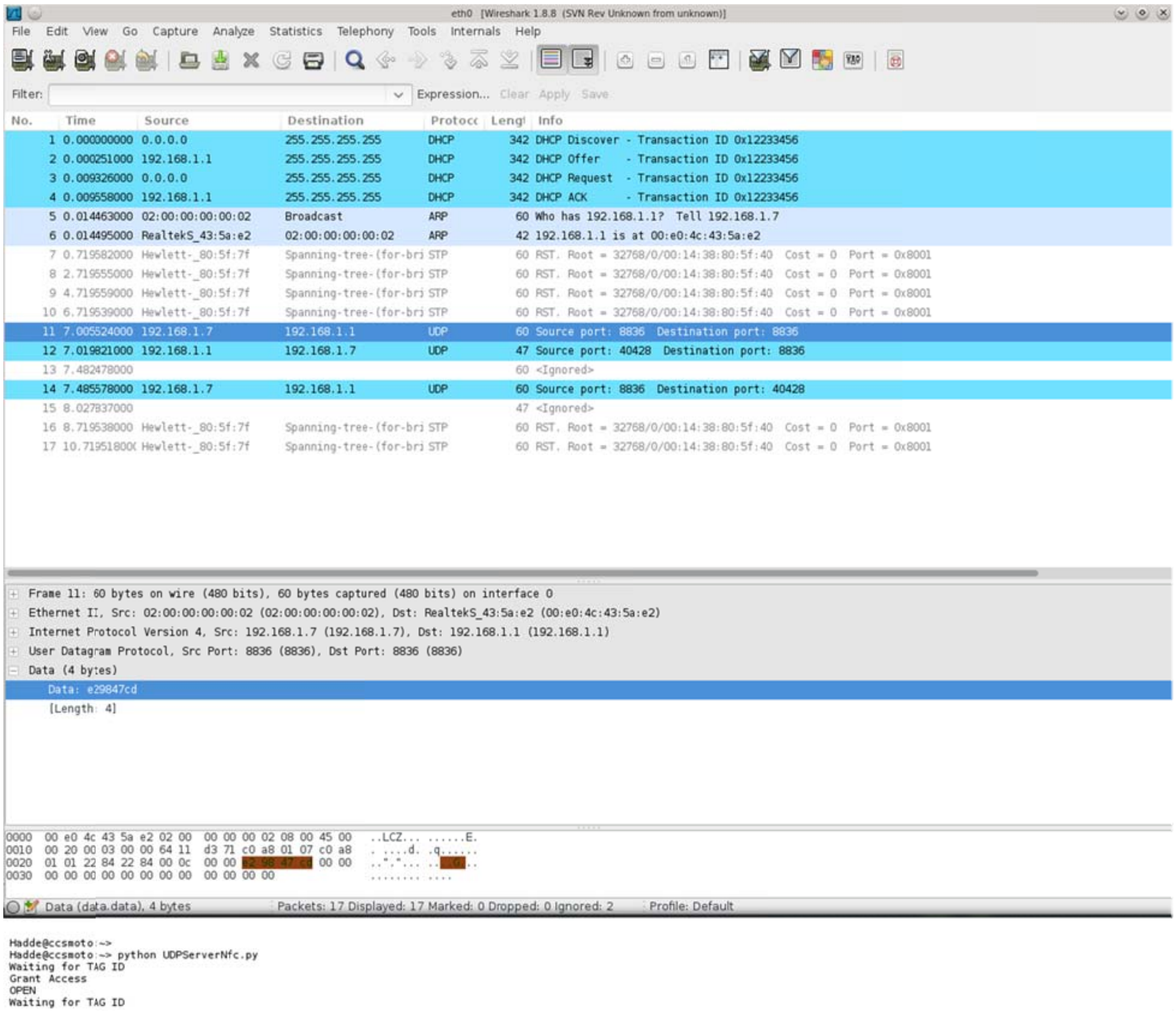
The UDP server program was written in the programming language Python [47] because of the simplicity of writing the code. The server program also generates console output indicating if the scanned NFC tag is to granted or denied access followed by an OPEN or CLOSE message being sent back to application running on the motherboard. For learning how to write a UDP server in Python we looked into the wiki page of Python for UDP communications [48]. When we had learned and understood how to create UDP sockets and transfer and receive UDP packets, we created our own very simple server which received different blocks of the NFC Tag UID. There were a total of 4 blocks received and each block represents 8bits. Once all the blocks are received from the NFC tag UID and converted from hexadecimal to decimal values we then utilized this value to lookup if this card should grant access or not. There are of course some flaws with this method since another Tag's UID might have the same value and then we would grant access even those we should not do so. However, this simple server was sufficient to verify that our UDP server combined with the motherboard reading the NFC Tag's UID was working.

The python server program is included in the appendix.

4.10 The Smart Door Lock

Now that everything is working, it was time for to test the whole system. This means that the board sends a UDP packet to the UDP (Python) server followed by receiving a UDP packet from the server which tells whether access was granted or denied. Finally the motherboard replies to the UDP server telling it if the lock has been opened or locked.

Figure 4-11 displays the output of Wireshark and the console output of the UDP server in the case when access is granted. We can see that a UDP packet was sent from the board to the UDP server which examines the tag's UID and the server's response UDP packet informing the board of its answer, followed by another UDP packet from the board informing the server if the lock is at that moment OPEN or CLOSED. In this example the tag will unlock the lock and grant access to the user. A similar procedure was used with a tag that was not registered with the UDP server resulted in a decision to deny access. We can see from these exchanges that the basic functionality of the smart door lock system has been realized.


eth0 [Wireshark 1.8.8 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.   Time         Source              Destination          Protocc Lengt Info
    1 0.000000000 0.0.0.0             255.255.255.255      DHCP     342 DHCP Discover - Transaction ID 0x12233456
    2 0.000251000 192.168.1.1         255.255.255.255      DHCP     342 DHCP Offer    - Transaction ID 0x12233456
    3 0.009326000 0.0.0.0             255.255.255.255      DHCP     342 DHCP Request  - Transaction ID 0x12233456
    4 0.009558000 192.168.1.1         255.255.255.255      DHCP     342 DHCP ACK      - Transaction ID 0x12233456
    5 0.014463000 02:00:00:00:00:02   Broadcast            ARP       60 Who has 192.168.1.1?  Tell 192.168.1.7
    6 0.014495000 RealtekS_43:5a:e2   02:00:00:00:00:02    ARP       42 192.168.1.1 is at 00:e0:4c:43:5a:e2
    7 0.719582000 Hewlett-_80:5f:7f   Spanning-tree-(for-bri STP     60 RST. Root = 32768/0/00:14:38:80:5f:40  Cost = 0  Port = 0x8001
    8 2.719555000 Hewlett-_80:5f:7f   Spanning-tree-(for-bri STP     60 RST. Root = 32768/0/00:14:38:80:5f:40  Cost = 0  Port = 0x8001
    9 4.719559000 Hewlett-_80:5f:7f   Spanning-tree-(for-bri STP     60 RST. Root = 32768/0/00:14:38:80:5f:40  Cost = 0  Port = 0x8001
   10 6.719539000 Hewlett-_80:5f:7f   Spanning-tree-(for-bri STP     60 RST. Root = 32768/0/00:14:38:80:5f:40  Cost = 0  Port = 0x8001
   11 7.005524000 192.168.1.7         192.168.1.1          UDP       60 Source port: 8836  Destination port: 8836
   12 7.019821000 192.168.1.1         192.168.1.7          UDP       47 Source port: 40428  Destination port: 8836
   13 7.482478000                                                    60 <Ignored>
   14 7.485578000 192.168.1.7         192.168.1.1          UDP       60 Source port: 8836  Destination port: 40428
   15 8.027837000                                                    47 <Ignored>
   16 8.719538000 Hewlett-_80:5f:7f   Spanning-tree-(for-bri STP     60 RST. Root = 32768/0/00:14:38:80:5f:40  Cost = 0  Port = 0x8001
   17 10.71951800( Hewlett-_80:5f:7f  Spanning-tree-(for-bri STP     60 RST. Root = 32768/0/00:14:38:80:5f:40  Cost = 0  Port = 0x8001
```

```
+ Frame 11: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
+ Ethernet II, Src: 02:00:00:00:00:02 (02:00:00:00:00:02), Dst: RealtekS_43:5a:e2 (00:e0:4c:43:5a:e2)
+ Internet Protocol Version 4, Src: 192.168.1.7 (192.168.1.7), Dst: 192.168.1.1 (192.168.1.1)
+ User Datagram Protocol, Src Port: 8836 (8836), Dst Port: 8836 (8836)
- Data (4 bytes)
    Data: e29847cd
    [Length: 4]
```

```
0000  00 e0 4c 43 5a e2 02 00  00 00 00 02 08 00 45 00   ..LCZ... ......E.
0010  00 20 00 03 00 00 64 11  d3 71 c0 a8 01 07 c0 a8   . ....d. .q......
0020  01 01 22 84 22 84 00 0c  00 00 e2 98 47 cd 00 00   ..".".... ...G...
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....
```

```
Data (data.data), 4 bytes      Packets: 17 Displayed: 17 Marked: 0 Dropped: 0 Ignored: 2      Profile: Default
```

```
Hadde@ccsmoto:~>
Hadde@ccsmoto:~> python UDPServerNfc.py
Waiting for TAG ID
Grant Access
OPEN
Waiting for TAG ID
```

Figure 4-11: **Tag with UID e29847cd is read and access is granted followed by response from the board.**

4.11 Power consumption

As we aimed to develop a cheap system, we have to consider the power consumption; the table below shows the power consumed by the different parts of the system running in different modes.

Table 4: **The power consumption of the different parts of the system**

Hardware	Active	Standby
MSP430f5437a	~ 0.0024 W*	~ 0.000004 W*
Motherboard	~ 0.96 W†	-
NFC Shield	~ 0.44 W†	~ 0.34 W†
Complete system	~ 1.4 W†	~ 1.3 W†

*This value is based on the information provided in the data sheet for the component, the highest values are used in the calculation.

† This measurement is based on the value shown on the Multimeter.

The system does not operate in the active state all the time. In active mode all the systems' functions are running, this includes the motherboard and all the components of the NFC Shield, as well as the electromagnetic field that the antenna produces to power the NFC tag. In standby mode the system does not send any packets through the Ethernet interface, also there is no NFC tag to provide with power.

The measurements are made the same way as Julia Alba Tormo Peiró describes in her master's thesis in section 3.2.1.1.6 [5].

5 Conclusions and Future work

This chapter summarizes our conclusions and suggests some future work that could be built upon what has been done and could address the parts of the original problem that have not been realized. The chapter ends with some reflections on the economic, social, and ethical issues considered during this thesis project.

5.1 Conclusions

To make a system that would achieve the project goal as stated in section 1.3 required that we achieve eight different sub goals (as stated that in the beginning of chapter 3). While we did not succeed in realizing some of our goals, we did managed to develop a working NFC reader that can read the UID of an NFC tag by using a NFC shield connected to a MSP430537a microcontroller via a SPI interface. We also managed to send UDP packets containing the NFC tag's UID to our UDP server that based upon the UID either granted or denied access to the user. Although the result of the access decision was indicated via a green or orange LED on the board, this could easily be turned into a signal to control a relay to activate an electric strike plate for a period of time or to active an alternative means of unlock/locking the door's lock.

Because the project spans of a wide range of disciplines involving both hardware and software, there were a lot of tools for us to learn and use. Programming a microcontroller and learning how to connect two different SPI devices included some debugging which led us to learn how to use a mixed signal oscilloscope. Also, we learned how to create, send, and receive UDP packets with the microcontroller. In addition to using the microcontroller we learned how to send and receive UDP packets using Python programming to realize our UDP server.

For monitoring the network traffic we used the software Wireshark. We learned how to make use of this software both for examining the protocols and packets that were being exchanged. Additionally, we learned that we could generate UDP messages within the application running on the microcontroller to display debugging information via Wireshark.

If we were to do the project again one of the things we would do differently is to read more about what is required of the components to function together *before* starting our coding. We would also not take for granted that things will work correctly right away. Furthermore, we will be more aware that there might be information missing in the documentation of the product, such as we experienced with the need for the second power source for the TXB0104PWR chip on the NFC shield. We think that reading the documentation and looking at the schematics of the components would prevent errors such as this from taking as much time as it did.

Additionally, rather than getting stuck on one problem in the early stage of the project one should focus on going further with the other goals rather than spending too much time attempting to fix one specific problem. In the end we managed to get a working system, despite some parts of the project being modified or eliminated because of the limited time for this project. For the future we learned that when developing a system we should do a lot more research (specifically reading and studying the documentation) in order to avoid minor problems causing a lot of frustration and unnecessary time being spend on them.

The most significant outcome of this project was that we learned how to interface the NFC shield to any SPI capable microcontroller, rather than requiring that the NFC shield be plugged into an Arduino. When doing research on this project we found that no one had previously solved this problem despite many people having encountered problems when trying to make use of NFC shields with platforms other than the Arduino. We published our solution to this in a posting[*] to the TI E2E Community on October 17 2013 at 06:56 AM so that others could build upon our solution.

[*] http://e2e.ti.com/support/microcontrollers/msp430/f/166/p/292483/1020503.aspx

5.2 Future work

Currently we have a NFC reader that only reads a NFC tag's UID then sends this information in a UDP packet to a simple UDP server that makes a decision and responds with an access granted or denied response. Our NFC reader is currently only reading a tag in MiFare target id mode; the next step would be to make it work in peer to peer mode so it can communicate with a smartphone with an NFC interface. Once the NFC reader works in peer to peer mode, then one could develop an Android application that will use NDEF for transferring a message. On the 31st of October 2013 Google introduced KitKat 4.4 [49], a version of Android which makes it possible for an Android smartphone with NFC to emulate a NFC smart card. This was something that we did not have during our project. This card emulation mode will be very helpful for future communication between the NFC reader and a smartphone.

Creating a full-fledged server with a database of all the NFC tag UIDs which should be granted access should be developed together with a webpage to manage the keys, update the database, and generate one-time or limited use soft tags that could be used by a smart phone – rather than being limited to making an access or deny decision based simply upon a tag's UID.

Because of the limited time we did not research how the lock should be installed on the door. The system needs to have a set of sensors and motors to be able to detect if a door is opened/closed and locked/unlocked, what type of sensors and motors should be used and how they should be installed is something that needs to be researched and developed.

An additional area that has not been explored in this thesis project is how to implement appropriate security for this system. This includes how to realize the appropriate cryptographic functions in the motherboard and how to realize them at the server. Using these cryptographic functions is should be possible to secure the boot loader (so that only properly signed code could be loaded), so that cryptographic tokens could be utilized for access control (rather than using UIDs which could easily be generated using a NFC equipped smartphone), and developing timed challenge response security so that one could ensure that the NFC device is actually in front of the NFC reader and not elsewhere (in order to avoid relaying of NFC communication). Also in the case of NFC tags there is unused memory in them which could be used to put in some security functions. Since our NFC reader forwards data to a UDP server there is a need of securing this transportation over the network, and using a usual Transport Layer Security might be a good idea to look into.

Last but not least it would be great to make the network bootloader work, since it would ease the process of distributing updates to the software that should be run in the board, while avoiding the need to physically attach the FET debugger to the board.

5.3 Reflections

The goal with our bachelor's thesis project was to simplify a way to control access via a locked door. Since NFC technology is becoming more widespread - as seen in public transportation, loyalty cards, and smartphones - the choice of using the NFC technology together with the UDP packets seems a very smart and useful tool for the future. In this sense we consider this bachelor's thesis project to have a positive social impact if this solution can be realized and deployed in the future.

The use of the MSP430F5437a ultra low power microcontroller together with NFC shield and having everything powered via PoE makes the hardware of this project energy efficient, this means environmental considerations were taken in the choice of the hardware. Moreover, because the board we used supported PoE all of the power for the system was sent over the Ethernet cable, thus eliminating the materials and labor costs associated with having to provide mains or another separate wiring plant to power the system.

As our board utilized the NFC shield designed for an Arduino by interconnecting the shield via an SPI interface with a MSP430F537a microcontroller with an ENC28J60 for its network communication, we avoided the need for an Arduino with two shields – one for NFC and for Ethernet and an external power supply and a nearby mains power outlet, thus there was a lower economic cost by using this hardware.

We have not encountered any ethical issues when carrying out this bachelor’s thesis project other than the issue of identifying users to be given access to a door based upon the UID of their NFC tag. This requirement could be removed in a future implementation that used cryptographic tokens. As noted in the previous section all security considerations have been ignored in this project and remain for future work.

References

[1] T. V. A. Pham, 'Security of NFC applications', Master's thesis, Royal Institute of Technology, School of Information and Communication Technology, Stockholm, Sweden, June 2013, TRITA-ICT-EX;2013:125, Available at http://kth.diva-portal.org/smash/record.jsf?searchId=2&pid=diva2:634369.

[2] G. Talaganov, '*Green VoIP : A SIP Based Approach*', Master's thesis, Royal Institute of Technology, School of Information and Communication Technology, Stockholm, Sweden, 2012, TRITA-ICT-EX;2012:162. Available at http://kth.diva-portal.org/smash/record.jsf?pid=diva2:539142.

[3] A. Lopez Garcia and F. J. Sanchez Galisteo, "*Exploiting wireless sensors: A gateway for 868 MHz sensors*", Master's thesis, Royal Institute of Technology, School of Information and Communication Technology, Stockholm, Sweden, Jun. 2012, TRITA-ICT-EX; 2012:110. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-98209 [accessed July 1, 2013].

[4] J. Lara Peinado, *"Minding the spectrum gaps : First steps toward developing a distributed white space sensor grid for cognitive radios"*, Master's thesis, Royal Institute of Technology, School of Information and Communication Technology, Stockholm, Sweden, 2013, TRITA-ICT-EX;2013:102. [Online]. Available at http://kth.diva-portal.org/smash/record.jsf?pid=diva2:627307

[5] T. P. J. Alba, 'Spectrum sensing based on specialized microcontroller based white space sensors : Measuring spectrum occupancy using a distributed sensor grid', Master's thesis, Royal Institute of Technology, School of Information and Communication Technology, Stockholm, Sweden, 2013, TRITA-ICT-EX; 2013:177. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?searchId=1&pid=diva2:636617. [Accessed: 11-November-2013].

[6] 'Lockitron'. [Online]. Available: https://lockitron.com/preorder. [Accessed: 01-July-2013].

[7] 'Yale unveils NFC door locks - NFC World', *NFC World*. [Online]. Available: http://www.nfcworld.com/2011/09/09/39785/yale-unveils-nfc-door-locks/. [Accessed: 04-July-2013].

[8] 'Vingcard debuts invisible door lock - NFC World', *NFC World*. [Online]. Available: http://www.nfcworld.com/2012/07/06/316689/vingcard-debuts-invisible-door-lock/. [Accessed: 04-July-2013].

[9] 'European Institute of Innovation and Technology – Innovative NFC access control solution from Telcred opens the doors to EIT ICT Labs Stockholm Co-location Centre'. [Online]. Available: http://eit.europa.eu/newsroom-and-media/article/innovative-nfc-access-control-solution-from-telcred-opens-the-doors-to-eit-ict-labs-stockholm-co-loc/. [Accessed: 04-July-2013].

[10] 'List of NFC phones'. [Online]. Available: http://www.nfcworld.com/nfc-phones-list/. [Accessed: 01-July-2013].

[11] 'Tablet • NFC phones and other devices • NFC World'. [Online]. Available: http://www.nfcworld.com/nfc-data/tablet/. [Accessed: 01-July-2013].

[12] 'npelly_jham_how-to-nfc_FINAL - how_to_nfc.pdf'. Available at http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/sv//events/io/2011/static/presofiles/how_to_nfc.pdf, [accessed July 1, 2013].

[13] 'What is PayPass NFC? | MasterCard®'. [Online]. Available: http://www.mastercard.com/us/paypass/phonetrial/whatispaypass.html. [Accessed: 01-July-2013].

[14] 'Visa payWave for Mobile'. [Online]. Available: https://developer.visa.com/paywavemobile. [Accessed: 01-July-2013].

[15] 'Visa Teams Up With Samsung on Contactless Mobile Payments - Bloomberg'. [Online]. Available: http://www.bloomberg.com/news/2013-02-25/visa-teams-up-with-samsung-on-contactless-mobile-payments.html. [Accessed: 01-July-2013].

[16] 'Alternatives for Banks to Offer Secure Mobile Payments.doc - MObey Forum 3 - 2010 - Alternatives for Banks.pdf'. Available at

https://www.nacha.org/userfiles/File/The_Internet_Council/Resources/MObey%20Forum%203%20-%202010%20-%20Alternatives%20for%20Banks.pdf, [accessed July 4, 2013].

[17] S. Kabir, "*Säkerhetsstandard för ett mjukvarubaserat säkerhetselement*." Bachelor's thesis, Royal Institute of Technology, School of Information and Communication Technology, Stockholm, Sweden, 2013, TRITA-ICT-EX; 2013:136. [Online] Available at http://kth.diva-portal.org/smash/record.jsf?searchId=1&pid=diva2:631622

[18] 'The TCP/IP Guide - TFTP Overview, History and Standards'. [Online]. Available: http://www.tcpipguide.com/free/t_TFTPOverviewHistoryandStandards.htm. [Accessed: 01-July-2013].

[19] 'IEEE-SA -IEEE Get 802 Program - 802.3: Ethernet'. [Online]. Available: http://standards.ieee.org/about/get/802/802.3.html. [Accessed: 15-November-2013].

[20] 'What is Power over Ethernet 102909.ppt - introduction_to_poe_ieee802.3af_802.3at.pdf'. Available at http://www.ieee.li/pdf/viewgraphs/introduction_to_poe_ieee802.3af_802.3at.pdf, [accessed July 1, 2013].

[21] 'What does the new Power over Ethernet standard mean for IT pros? - TechRepublic'. [Online]. Available: http://www.techrepublic.com/blog/data-center/what-does-the-new-power-over-ethernet-standard-mean-for-it-pros/. [Accessed: 29-November-2013].

[22] 'Code Composer Studio (CCStudio) Integrated Development Environment (IDE) v5 - CCSTUDIO - TI Tool Folder'. [Online]. Available: http://www.ti.com/tool/ccstudio. [Accessed: 15-November-2013].

[23] '54645D 100-MHz 200-MSa/s Mixed Signal Oscilloscope [Obsolete] | Agilent'. [Online]. Available: http://www.home.agilent.com/en/pd-1000001438%3Aepsg%3Apro-pn-54645D/100-mhz-200-msa-s-mixed-signal-oscilloscope?&cc=SE&lc=eng. [Accessed: 15-November-2013].

[24] 'MSP430 USB Debugging Interface - MSP-FET430UIF - TI Software Folder'. [Online]. Available: http://www.ti.com/tool/msp-fet430uif. [Accessed: 08-November-2013].

[25] 'Arduino - ArduinoBoardUno'. [Online]. Available: http://arduino.cc/en/Main/arduinoBoardUno. [Accessed: 08-November-2013].

[26] 'ATmega48PA/88PA/168PA/328P - doc8161.pdf'. Available at http://www.atmel.com/Images/doc8161.pdf, [accessed November 8, 2013].

[27] 'ProCurve Switch 2626 (J4900B) specifications - HP Products and Services Products'. [Online]. Available: http://h10010.www1.hp.com/wwpc/ca/en/sm/WF06b/12136296-12136298-12136298-12136298-12136316-12136318-31539227.html?dnr=2. [Accessed: 20-November-2013].

[28] 'Drivers & Downloads | Dell US'. [Online]. Available: http://www.dell.com/support/drivers/us/en/19/Product/optiplex-gx620. [Accessed: 20-November-2013].

[29] 'openSUSE'. [Online]. Available: http://en.opensuse.org/Main_Page. [Accessed: 07-November-2013].

[30] 'MSP430F543xA, MSP430F541xA Mixed Signal Microcontroller (Rev. C) - msp430f5437a.pdf'. Available at http://www.ti.com/lit/ds/symlink/msp430f5437a.pdf, [accessed July 29, 2013].

[31] 'ENC28J60 Data Sheet - 39662c.pdf'. Available at http://ww1.microchip.com/downloads/en/DeviceDoc/39662c.pdf, [accessed July 29, 2013].

[32] 'TL2575, TL2575HV 1-A Simple Step-Down Switching Voltage Regulators (Rev. B) - tl2575hv-adj.pdf'. Available at http://www.ti.com/lit/ds/symlink/tl2575hv-adj.pdf, [accessed July 29, 2013].

[33] 'TPS237x: IEEE 802.3af PoE Powered Device Controllers (Rev. B) - tps2375.pdf'. Available at http://www.ti.com/lit/ds/symlink/tps2375.pdf, [accessed July 29, 2013].

[34] 'NFC Shield [SLD80453P] - $29.50 : Seeed Studio Bazaar, Boost ideas, extend the reach'. [Online]. Available: http://www.seeedstudio.com/depot/nfc-shield-p-916.html. [Accessed: 29-July-2013].

[35] 'TXB0104 4-Bit Bidirectional Voltage-Level Translator (Rev. F) - txb0104.pdf'. Available at http://www.ti.com/lit/ds/symlink/txb0104.pdf, [accessed July 29, 2013].

[36] 'Internet Systems Consortium | DHCP'. [Online]. Available: http://www.isc.org/downloads/dhcp/. [Accessed: 26-July-2013].

[37] ‘Portal:YaST - openSUSE’. [Online]. Available: http://en.opensuse.org/Portal:YaST. [Accessed: 07-November-2013].
[38] ‘Tutorials | Gustavo Litovsky’. [Online]. Available: http://glitovsky.com/blog/?page_id=21. [Accessed: 29-July-2013].
[39] ‘SPI Interface in embedded systems’. [Online]. Available: http://www.eeherald.com/section/design-guide/esmod12.html. [Accessed: 29-July-2013].
[40] ‘Oracle VM VirtualBox’. [Online]. Available: https://www.virtualbox.org/. [Accessed: 07-November-2013].
[41] ‘User Manual - 141520.pdf’. Available at http://www.nxp.com/documents/user_manual/141520.pdf, [accessed July 29, 2013].
[42] ‘Energia Reference - Introduction’. [Online]. Available: http://energia.nu/. [Accessed: 20-November-2013].
[43] ‘Voltage Level Translation - Dual Supply Translator - TXB0104 - TI.com’. [Online]. Available: http://www.ti.com/product/txb0104. [Accessed: 15-November-2013].
[44] ‘MIFARE Classic :: NXP Semiconductors’. [Online]. Available: http://www.nxp.com/products/identification_and_security/smart_card_ics/mifare_smart_card_ics/mifare_classic/. [Accessed: 07-November-2013].
[45] ‘NFC TagWriter by NXP - Android-appar på Google Play’. [Online]. Available: https://play.google.com/store/apps/details?id=com.nxp.nfc.tagwriter&hl=sv. [Accessed: 07-November-2013].
[46] ‘Home :: NXP Semiconductors’. [Online]. Available: http://www.nxp.com/. [Accessed: 07-November-2013].
[47] ‘Python Programming Language – Official Website’. [Online]. Available: http://www.python.org/. [Accessed: 07-November-2013].
[48] ‘UdpCommunication - Python Wiki’. [Online]. Available: https://wiki.python.org/moin/UdpCommunication. [Accessed: 20-November-2013].
[49] ‘Android KitKat | Android Developers’. [Online]. Available: http://developer.android.com/about/versions/kitkat.html. [Accessed: 15-November-2013].

Image Reference

Figure 3-5: Wikipedia, URL: http://en.wikipedia.org/wiki/File:SPI_timing_diagram2.svg Retrieved: 2013-10-10

Appendix

Source Code

All the files and source codes involved in our bachelor project are available in this Dropbox folder https://www.dropbox.com/sh/43h519564hspkw2/RcXkXGtCqQ. The subfolders in this main folder link to different sections in our project.

- The DHCP CONFIG folder contains the DHCP server configuration we used in the openSUSE PC so that the board could connect to it and receive an IP.
- The UDP Server folder contains the necessary UDP server code in the language Python which received and responded with UDP packets to our board.
- The UDPSmartLock contains all the necessary code for our board in order to read NFC UID Tags and also send/receive UDP packets.

DHCP server CONFIG

```
option tftp-server-address code 150 = ip-address;
option tftp-server-address 192.168.1.1;
option bootfile-name "frequencyScanner.txt";
option boot-size 76;
ddns-update-style none;
default-lease-time 86400;
max-lease-time 86400;
authoritative ;
subnet 192.168.1.0 netmask 255.255.255.0 {
  option subnet-mask 255.255.255.0;
  option broadcast-address 192.168.1.255;
  option routers 192.168.1.2;
  range 192.168.1.5 192.168.1.20;
  host NFC-Shield {
    hardware ethernet 02:00:00:00:00:02;
    fixed-address 192.168.1.7;
  }
  host Scannerboard {
    hardware ethernet 02:00:00:00:00:03;
    fixed-address 192.168.1.5;
  }
}
```

UDP Python Server

```
from socket import *
import sys
import select
import ast
import os

#address of the UDP server and port
address = ('192.168.1.1', 8837)

#create socket for internet and UDP
server_socket = socket(AF_INET, SOCK_DGRAM)
#bind this socket to our address
server_socket.bind(address)

while(1):

    print "Waiting for TAG ID "
    #buffer size is 65535 bytes (random choice)
    recv_data, addr = server_socket.recvfrom(65535)
    #first block (8bits) of received message
    int1=recv_data[0]
    #second block (8bits) of received message
    int2=recv_data[1]
    #third block (8bits) of received message
    int3=recv_data[2]
    #fourth block (8bits) of received message
    int4=recv_data[3]
    #convert blocks from hexadecimal to decimal
    intsum1=ord(int1)
    intsum2=ord(int2)
    intsum3=ord(int3)
    intsum4=ord(int4)
    #summarize the values
    intsum = (intsum1+intsum2+intsum3+intsum4)

#in this case the value 654 represents the tag UID e29847cd  in decimal
    if 654 == intsum:
    server_socket1 = socket(AF_INET, SOCK_DGRAM)
        data1 = 'OPEN'
        print "Grant Access"
     #if granted then send 'OPEN' to our board
    server_socket1.sendto(data1, addr)

    elif recv_data != intsum:
    server_socket2 = socket(AF_INET, SOCK_DGRAM)
        print "Denied"
        data2 = "CLOSE"
      #if denied then send 'CLOSE' to our board
    server_socket2.sendto(data2, addr)
```

UDPSmartLock critical functions

```
//Spi for UCA1
uint8_t spi_send(const uint8_t _data) {
            while (!(UCA1IFG & UCTXIFG))
                        //UC0IFG & UCB0TXIFG
                        ;// wait for previous tx to complete

            UCA1TXBUF = _data; // setting TXBUF clears the TXIFG flag

            while (!(UCA1IFG & UCRXIFG))
                        ; // wait for an rx character?

            return UCA1RXBUF; // reading clears RXIFG flag
}


//SS, shifts the output on SS pin (pin5)
void digitalWrite(uint8_t val) {
            if (val == LOW) {
                        P5OUT &= ~BIT5;
                        P5DIR = BIT5; // set bit 5 to output
            } else {
                        P5OUT |= BIT5;
                        P5DIR = BIT5; // set bit 5 to output
            }
}

/*Function:Transmit a byte to PN532 through the SPI interface. */
inline void write(uint8_t _data) {
            spi_send(_data);
}

/*Function:Receive a byte from PN532 through the SPI interface */
inline uint8_t read0(void) {
            uint8_t data_ = spi_send(0);
            return data_;
}
```

These functions are the necessary functions to make example code from SeeedStudio to work, the example code can be found on http://www.seeedstudio.com/wiki/File:PN532_SPI_V2.zip

TRITA-ICT-EX-2013:257