Masterarbeit

Development of an XML Description for Secure Embedded Systems based on NFC

Peter Tielsch, BSc

Institut für Technische Informatik
Technische Universität Graz
Vorstand: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Römer

Begutachter:	Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger
Betreuer:	Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger
	Dipl.-Ing. BSc Manuel Menghin

Graz, Oktober 2013

Kurzfassung

Eingebettete Systeme sind heutzutage aus dem Alltag kaum mehr wegzudenken. Da auch die Nahfeldkommunikation (NFC) zu einer immer beliebteren und weitgehend verfügbaren Technologie wird, entsteht so die Idee eingebettete Systeme mittels NFC-fähigen Smartphones zu steuern. Einige der daraus resultierenden Vorteile für den Benutzer sind sowohl eine einheitliche Benutzerschnittstelle als auch mehr Kontrolle über das eingebettete System (z.B. administrativer Zugriff, Firmware Updates). Diese Arbeit beschäftigt sich mit der Entwicklung einer XML-basierten „descriptive markup language", die es ermöglicht die Funktionalität von gängigen elektronischen Geräten wie einer Mikrowelle oder einer Waschmaschine zu beschreiben sowie deren Integration in das bereits existierende NFC Interface System, einer NFC-fähigen Lösung für eingebettete Systeme, ermöglicht. Die Beschreibung der Funktionalität dient des Weiteren der Erstellung einer dynamischen Benutzeroberfläche, welche den vollen Zugriff auf das eingebettete System erlaubt. Besonderes Augenmerk liegt auf einem Konzept, dass den Zugriff auf bestimme Geräte Funktionalität nur autorisierten Benutzern erlaubt, zum Beispiel Angestellten des Verkäufers, um das Gerät zu warten oder zu konfigurieren. Weiters wird eine Android App entwickelt, welche die Möglichkeit bietet mit dem Gerät anhand der Funktionalitäts-Beschreibung zu kommunizieren und zu interagieren, und so eine generische Kontrolle von elektronischen Geräten mittels eines NFC-fähigen Android Smartphones erlaubt. Anhand einer Fallstudie für zwei konkrete Anwendungsfälle, einer Mikrowelle und einem Smart Meter, wird zuletzt die Funktionalität des implementierten Systems analysiert.

Abstract

Nowadays, everyday life is barely imaginable without embedded systems. As Near Field Communication (NFC) becomes an increasingly available and popular technology, the idea of interfacing embedded systems with modern NFC-enabled smartphones emerges. Some of the immediate advantages for the user are unified user interfaces as well as gaining more control over the systems (e.g. admin access, firmware updates,...). This thesis deals with the development of an XML-based "descriptive markup language" with the potential to describe the functionality of popular everyday embedded systems like microwave ovens or washing machines as well as its integration into the existing NFC Interface system, an NFC-enabled solution for embedded systems. The functionality description will then be used in order to create a dynamic user interface allowing full access to the embedded system's functionality. Special focus lies on a concept to allow access to certain device functionality only to authorized users, like vendor staff for maintaining or configuring devices. Furthermore, an Android app being able to communicate and interact with the system according to the device's functionality description will be developed, thus introducing a generic way to control consumer electronic devices via Android NFC-enabled smartphones. A case study is finally conducted for two specific use cases, a microwave oven and a smart meter, in order to evaluate the functionality of the implemented system.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Danksagung

Diese Diplomarbeit wurde im Jahr 2013 am Institut für Technische Informatik an der Technischen Universität Graz in Kooperation mit Infineon Technologies Austria AG durchgeführt.

Ich möchte mich für die Unterstützung vom Institut für Technische Informatik durch Ass.Prof.Dipl.-Ing.Dr.techn. Christian Steger und insbesondere Dipl.-Ing. Manuel Menghin, BSc während der Masterarbeit bedanken. Weiters danke ich Infineon Technologies Austria AG und Dr. Josef Haid für die Möglichkeit der Durchführung eines äußerst interessanten und vielseitigen Praktikums während der Masterarbeit.

Mein Dank gilt ausserdem meiner Familie und all meinen Freunden für die Unterstützung während meines Studiums.

Graz, im Oktober 2013 Peter Tielsch

Contents

List of Figures

List of Tables

List of Code Examples

Chapter 1

Introduction

1.1 Motivation

In 1991, Mark Weiser described what he called „Ubiquitous computing“ as his idea of how computers would integrate into everyday live in the future. His vision includes an ever growing amount of devices, including sensors, tablet-like handheld devices and large screens gaining an ever growing importance on our daily lives. Furthermore, he suggested that the importance of these devices would not be derived from their individual functionality, but from their interaction. Weiser and his team developed a device called „Active Badge“ which allowed the wearer to be identified and located within a building, which allows the automatic opening of doors, forwarding of telephone calls and automatic logon to computer terminals. [Wei91]
Today, consumer electronic devices (aka. embedded systems) are all around us, every modern kitchen and living room is filled with them. Such devices conveniently offer users profit from a variety of functions which had been unthinkable not so long ago. Another step towards making those systems more convenient for users is to offer them a single interface for the functionality every one of their systems has to offer. What would be better suited for this than a smartphone? Most users already own one, so it's a familiar environment and there is no need of getting accustomed to a new kind of device or user interface. A number of technologies qualify for this task as they are widely supported by modern smartphones. Some of these technologies are Bluetooth[1], wireless LAN[2] and Near Field Communication[3] (NFC). This thesis focuses on achieving such an interface via NFC technology.

[1] http://www.bluetooth.com
[2] http://www.ieee802.org/11/
[3] http://www.nfc-forum.org/

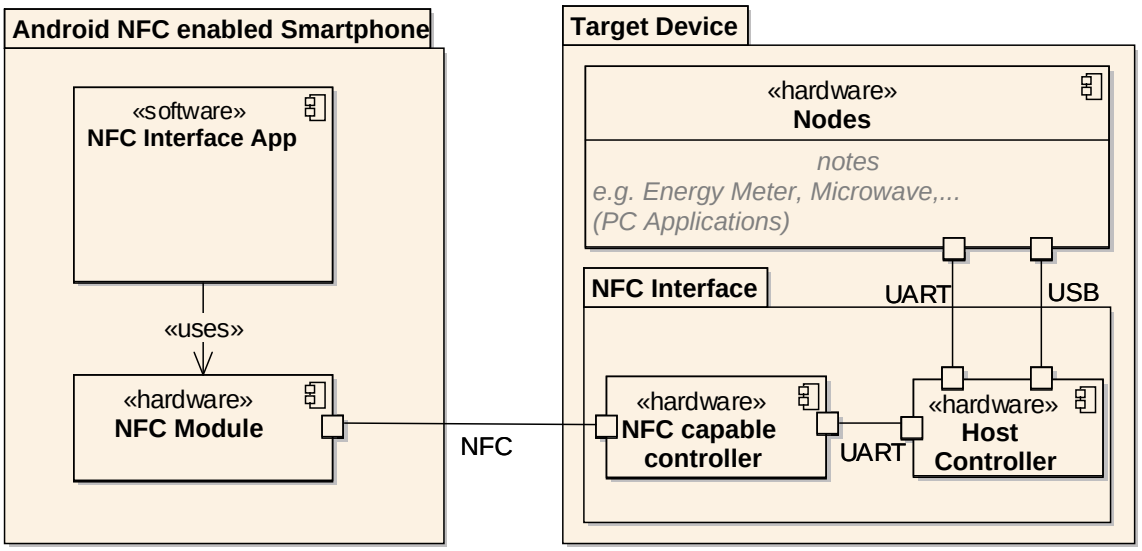

Figure 1.1: Architecture of the current NFC Interface system, including the Target Device and the Android based NFC Reader.

1.2 Current State

This thesis is part of the META[:SEC:][4] project, being conducted at the Institute for Technical Informatics in cooperation with Infineon Technologies Austria AG[5] as an industrial partner. The META[:SEC:] project conducts research and development in the field of contactless NFC systems and smart cards with special interest in power and fault awareness as well as security.
This project builds upon an existing NFC Interface system (NFCI) described by[Bas13] and [DM12], which so far allows electronic devices to be equipped with an NFC communication interface. This further on allows the device to be controlled by a modern NFC-enabled smartphone. The smartphone has to be equipped with an app allowing communication with and control over the device. The Target Device consists of multiple hardware components:

- An NFC capable controller including the NFC circuitry
- A host controller allowing communication with the nodes
- The nodes, which could be any kind of electronic devices

The current system is depicted in Figure 1.1.

Compared to other technologies, the usage of NFC has many perks, for example that there is no need for pairing devices (Bluetooth) or connecting to a network (WLAN). Also, the

[4]Mobile Energy-efficient Trustworthy Authentication Systems with Elliptic Curve based SECurity; funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract 829586.
[5]http://www.infineon.com/austria

costs of one NFCI system prototype run below four Euros, which already makes it a very cheap solution as concluded by [Bas13]. Another unique feature of the NFCI system is the so called Zero Energy Standby described by [DM12]. It allows for a complete power down of the Target Device circuitry, and the activation of the power supply upon the NFCI retrieving power through the electromagnetic field, thus leading to far less standby energy consumption.

Further work has already been conducted in order to enable secure communication between the NFC-enabled Android smartphone and the Target Device, including the communication path to the nodes using state of the art ECC and AES technology. [Fio13]

The nodes can be thought of as hardware components containing the actual functionality an end user would be interested in. For example, in an access control system one node might be the number pad while a second one might be a fingerprint scanner and a third one the door lock. In the current system, the nodes attached to the NFCI are emulated by PC applications as these are easier to maintain during the development process than real hardware components.

A limitation the system faces in its current state is the NFC-enabled smartphone user interface for interfacing the Target Device. It is very simplistic and requires all Target Device functionalities to be known in advance. In case of changes to the Target Device (e.g. a new node), the smartphone app needs to receive an update in order to function properly with the Target Device. In the long term, this could lead to horrendous backward compatibility code as well as very frequent updates for hardcoded functionality.

1.3 Goals

The goal of this project is to solve the issue of communication and interaction between two devices, an NFC reader (Android smartphone) and an embedded device featuring the NFCI system unknown to each other. The following list of goals specifies the concrete desired results in more detail:

1. Specification of a markup language being able to describe any functionality the Target Device, consisting of the NFCI as well as its attached nodes, has to offer.

2. Development of an Android app being able to interpret the Target Device Description (TDD) and create a graphical user interface that allows interaction with the Target Device's functionality. Together with Task 1 these are the main goals.

3. Extending the NFCI firmware. The Target Device shall store its functionality description and be able to provide it to Android NFC-enabled smartphones with the NFCI app installed. The description shall be stored in the NDEF message format (see [For06a] for the full specification) and shall be retrieved using the TagType4 standard (see [For11] for the full specification).

4. Extension of the existing Target Device PC applications in order to support all new features and implementations.

5. Implementation of test cases in order to help maintain the system in the future, and automatically test any part of the Android app without the need for user interaction (e.g. for testing the app's NFC communication).

6. A case study shall be conducted to evaluate the markup language and the required implementations of all system components.

Chapter 2

Related Work

The related work shall give an overview of the technological baseline for this project. As the whole system is based on NFC technology, any aspects related to NFC such as how NFC can help in improving user interaction, usefulness and other innovative use cases shall be looked at first, followed by an analysis of the potential of interfacing embedded systems via NFC. As the goal of this project is to find and/or implement a language capable of describing the functionality of the NFC Interface (NFCI) system (see chapter 1.3), the next part of the related work will deal with existing languages aimed at solving similar problems. They will be divided into two categories. These categories are user interface (UI) description languages and languages describing small hardware devices (sensors, actors) such as in home automation or sensor network environments. Finally, the viability of using XML as the base technology in order to create a language that is able to describe the NFCI system functionality shall be evaluated.

2.1 Near Field Communication

As the NFCI system this project builds upon (see chapter 1.2) uses Near Field Communication (NFC) as its main communication path, this section shall give a short overview of the technology, its restrictions, current use cases and as well investigate potential future improvements for the NFCI through innovative usage of the technology.

2.1.1 Technical Overview

Near Field communication is a wireless radio communication technology for electronic devices over short distances. It is based on RFID technology and allows users to commit intuitive contactless actions via their electronic, NFC ready, devices. NFC tags are passively powered devices receiving their energy only through the electromagnetic field generated by an active NFC device. They can be used to store information to be retrieved by an active NFC device. The maximum memory as well as other technical details like maximum speed (which ranges between 106 and 424 kbit/second) depend on the NFC tag type, which is a set of standards defined by the NFC Forum. The source [RG] provides a detailed technical overview of the NFC technology beyond the short introduction given here. [For08] provides an overview of NFC as an ecosystem including the role of NFC-enabled smartphones and other NFC capable hardware components.

NFC uses the so-called NDEF message ("NFC Data Exchange Format") standard for the exchange of larger binary data. This format has been defined by the NFC Forum and is the current standard used for Near Field Communication. NDEF is a lightweight binary format, including a header that allows application specific data to be typed as URI, MIME media type (e.g. "image/jpeg") and others, if needed for application independent usage. An exact specification of the NDEF message format can be found at [For06a].

2.1.2 Security

The authors of [HB06] have evaluated security in Near Field Communication systems. They identified a number of possible threats as well as solutions to these threats. These threats are as follows:

- Eavesdropping: Due to the wireless nature of NFC, eavesdropping is an important issue. NFC itself cannot protect users against threat. The best solution to this problem is using a secure channel.
- Data Corruption: In this case, instead of just listening the attacker tries to invalidate the transmitted data. Data corruption can be achieved by transmitting valid frequencies of the data spectrum at the correct time. This attack is detectable by the victims as the power required to corrupt the data is significantly above the power level normally used for NFC-based data transmission.
- Data modification: In this case, the attacker wants to disturb communication, and in addition wants the victim to receive manipulated but valid data. While there are some solutions to this problem, the best one would be using a secure channel.
- Data insertion: In this case, the attacker inserts messages into the communication between two devices. This becomes possible should the attacker be able to respond faster than the attacked device. A secure channel provides protection from these types of attack as well.
- Man-in-the-Middle attack: This kind of attack is deemed practically infeasible by the authors. It requires the attacker to disturb transmissions which can be noticed by the victim.

The secure channel mentioned in the threat analysis above is the only way to effectively take care of any threats with a single solution. For two NFC devices, key agreement protocols like Diffie-Hellmann based on asymetric encryption can be used to establish a shared secret. Furthermore, due to Man-in-the-Middle attacks not being a threat in the first place, an unauthenticated version of the Diffie-Hellmann key agreement protocol would be enough. At this point, a symmetric key can be derived of the shared secret ensuring protection against all attacks analyzed above.

2.1.3 Use Cases

NFC is already applied in a variety of fields today. Some everyday use cases include:

Data exchange:

- Exchanging data between smartphones (e.g. contact information, pictures or music)[1] without the need for pairing (e.g. Bluetooth) or similar ways of establishing a connection, which makes NFC much easier to use compared to other technologies
- Enables the possibility of sending data to other smart devices (e.g. printers) for further processing
- Reading from and writing data to NFC tags (e.g. vcards)
- The Austrian Federal Railways "INFOStation" allows users to retrieve train schedules via NFC [OEB]

Mobile payment systems:

- Google wallet[2]: The basic idea of this safe payment system is to allow a user to pay in a shop without handing out his payment information. This information (e.g. a credit card) is only known to the Google wallet. It can then conduct the payment without handing over the credit card information. Google offers an Android app allowing usage of Google wallet as well for NFC-based payments.
- Austrian Federal Railways partially offer NFC payment for tickets.

2.1.4 Innovative User Interaction through NFC

Many researchers have already looked into the possibilities of using NFC to introduce new, intuitive and useful ways of user interaction. Their research mostly includes NFC tags and other NFC capable hardware and aims at introducing new ways of using mobile computing devices to simplify everyday live.

The paper [AV07] explains how NFC can be used to facilitate the usage of other networking technologies. For example, a connection to a wireless LAN can be established via touching a NFC tag containing information on how to establish the connection. This is specially advantageous to non-expert users who might not have the expertise to do this on their own. This example of usage benefits greatly from one of the unique features of NFC, which is the way it can express a users intent to interact with an object within physical vicinity. This is extremely intuitive as it follows the natural tendency of users controlling things by pointing at or physically touching them.

In [BHP+08] the authors describe the "Collect & Drop" technique, which allows interacting with real world objects. They are tagged with additional digital information such as a link to a web page by using NFC tags, or other technologies (e.g. QR codes). Upon a user

[1] http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#p2p
[2] https://wallet.google.com

discovering a tag, he is not prompted to immediately interact with it, there is no imposed order of interaction. Instead, information collected from tags is stored in so called Collections. A Collection is a container consisting of a URL to a service and required parameters. Upon a user deciding to open a Collection and interact with it, he can then choose from the given parameters and invoke the service. An example for the Collect&Drop system is a public transportation ticketing system. After retrieving the information from a tagged poster, the user can simply choose a starting point, a destination and the number of persons in order to complete the ticket purchase, without having to understand a complicated system.

The authors of [SRP09] describe an interaction model named "Touch & Compose" in a so called "smart environment", which defines an environment containing a large amount of interconnected sensors, devices and smart everyday objects. It is based on the "Collect & Drop" technique but refines it further. The basic idea is for the user being able to pick up resources during his everyday life by touching them (e.g. exhibits in a museum), and use them later on. In order to achieve natural interaction with smart tags, the authors introduce three phases of user interaction. The first one, the discovery stage, is where a visual scan can reveal tags in the room in which the user might be interested in, similar to a textual search. The user can also be guided towards the physical representation of the information he has collected so far. After the discovery stage, the user can utilize the natural way of touching objects he is interested in, in order to collect their information. This is called the composition phase. The third phase is the so called usage phase, which allows the user to use the collected information in combination with other smart devices (e.g. a museum exhibits contained a video, which can now be watched using a media player).

[CPL11] and the consecutive paper [CPL12] are as well dealing with the topic of intuitive user interaction via NFC. They describe an idea similar to [AV07], however they provide a concrete implementation using a modern Android smartphone. They put NFC tags on consumer electronic devices supporting wireless connections such as Wifi or Bluetooth. These tags contain information on the device's wireless connection and how to connect to the supported network. There are some implemented applications in the paper. For example, the "Touch & Listen" application can stream music from the smartphone directly to a Bluetooth speaker once the user touches the speaker's NFC tag. The "Touch & Watch" application allows streaming a video file to a television set in the same manner.

2.2 NFC Interfaces

A short introduction to the NFCI system in general has already been given in chapter 1.2. The exact details on how the communication protocol as well as the communication hardware components are implemented in order to allow communication between the NFC reader and the Target Device as well as between the Target Devices components itself will be given in this section. This has been defined by the thesis this projects builds upon [Bas13].

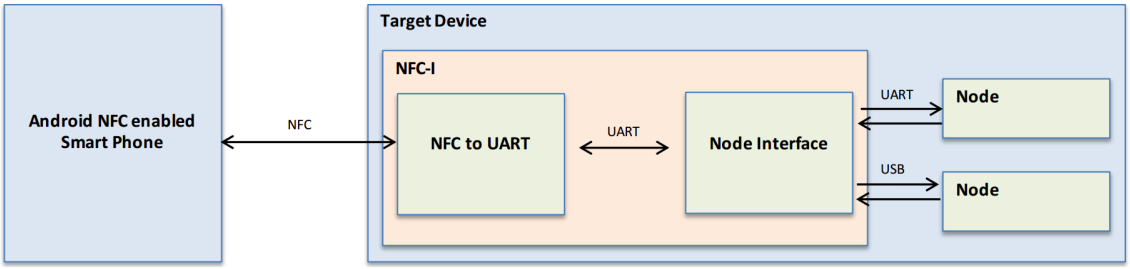

Figure 2.1: Communication path within the NFC Interface system

Figure 2.1 depicts all essential components required in order to control a node via the NFC reader. These components are:

- Android NFC-enabled smartphone: The device used to control the Target Device.
- NFC: An overview of NFC as a wireless communication technology can be found in chapter 2.1.
- NFC to UART: This component is part of the NFCI and is responsible for the contactless communication with the Android NFC-enabled smartphone.
- Node interface: This component is part of the NFCI as well and handles the contact-based communication between the NFCI and the Target Device nodes.

2.2.1 Communication Protocol

The NFC-based data exchange between the Android NFC-enabled smartphone and the Target Device is implemented via so called Application Protocol Data Unit (APDU) commands, which are basically byte arrays. Their structure is defined by the ISO/IEC 7816-3 standard.

There are two types of APDUs, the command APDU which is created by the Android NFC-enabled smartphone in order to send data to the Target Device and the response APDU, which is created by the Target Device and sent back to the Android NFC-enabled smartphone as a response to the incoming command.

2.2.2 Hardware

The NFCI features two hardware components (as depicted in 1.1) allowing communication with the NFC reader as well as the connected nodes.

The NFC to UART component is the master and initiates any communication with the node interface. It is powered through the magnetic field induced by the Android NFC-enabled smartphone, however the UART module it uses is additionally powered through the node interface. The whole communication between the node interface and the actual nodes is UART-based.

In the current conceptual system, the nodes are not yet actual electronic hardware devices, but are emulated by PC applications attached to the NFC to UART component via USB.

2.2.3 Security

The author of the thesis [Fio13] has recently introduced a security concept for the NFCI system allowing secure communication between the NFC-enabled Android smartphone and the Target Device, including an encrypted communication path to the nodes. The implementation uses the state of the art ECC-based Diffie Hellman key exchange protocol and AES-based symmetric encryption for exchanging data. Combined, these technologies will provide state of the art security to the NFCI system in the future.

2.3 Description Languages for User Interfaces

User interface (UI) description languages are used to describe UIs as well as the functionality attached to them. These languages are usually platform independent themselves, however, language interpreters are needed to display the described user interface in a certain environment. They have a lot in common with what is needed for the NFCI system as they offer a remote user interface for some kind of functionality.

Many languages couple the functionality of the described interface with its UI and are therefore not necessarily pure markup languages. In such cases, it might not be that easy to access functionality without using the predefined UI. Some languages like UIML2 separate the two clearly. This feature allows more flexible use of the described UIs, e.g. when multiple devices should be managed by an automated system instead of only being controlled through the UI itself.

2.3.1 Android User Interfaces

Android offers an XML-based UI description. This allows developers to design the graphical user interface (GUI) to be displayed on the screen in an easily readable, writeable and modifiable way. The XML resource file does not contain any functional code. Therefore it also offers a clear separation of UI design on the one hand and functional code based on it on the other.

Android XML-based User Interface Example

Code 2.1 Simple Android UI example

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3                android:layout_width="fill_parent"
4                android:layout_height="fill_parent"
5                android:orientation="horizontal" >
6      <EditText android:id="@+id/input"
7                android:layout_width="wrap_content"
8                android:layout_height="wrap_content"
9                android:text="Write something here" />
10     <Button android:id="@+id/button"
11             android:layout_width="wrap_content"
12             android:layout_height="wrap_content"
13             android:text="Click here" />
```

```
14 </LinearLayout>
```

Code 2.1 shows a very simple UI composed of three elements: an *EditText*, a *Button* and a *LinearLayout*. The *LinearLayout* simply aligns the widgets it contains according to the *android:orientation* attribute; in this case below each other. The *EditText* is able to show textual information to the user as well as allowing the user to provide text based input, while the *Button* will usually execute some kind of functionality on being interacted with.

Functionality can be attached to any element declared in the XML layout file in a simple way, as shown in Code 2.2. The code creates an *EditText* object based on the one declared in the XML file allowing it to be manipulated programmatically afterwards. In this example, the text being displayed on the screen is saved into a string variable.

Code 2.2 Modifying XML declared Android user interface components

```
1 EditText editText = (EditText) Activity.findViewById(R.id.input);
2 String text = editText.getText().toString();
```

Runtime UI Creation

Android XML layout files are compiled into view resources at compile time. The Android system does not support creation of these resources for an application at runtime. Therefore, such layouts cannot be loaded from external sources whenever necessary. However, it is possible to create *View* and *ViewGroup* objects at runtime programmatically, thus offering the same possibilities as with XML-based layouts. [3]

2.3.2 UIML2

The "User Interface Markup Language 2" (UIML2) is a markup language developed by Constantinos Phanouriou in 2000 at the Virginia Polytechnic Institute and State University as a successor to UIML. A detailed description of the standard is given in [Pha00].

UIML2 is an XML-based language with the goal to declare user interfaces in a way that allows different devices and applications to make use of them. On the target platform, the description has to be interpreted to actual elements of a supported UI toolkit (e.g. for the Java Swing UI Kit: javax.Swing.JFrame). Thus, the requirement for using UIML2 on a specific target platform is having an interpreter that understands the UIML2 language and is able to map it to UI elements used by the target platforms UI toolkit (e.g. Java SWING, Android UI).

UIML2 does not support programming of application logic; therefore the user interface described by UIML2 has to be connected to the required application logic.

Features of UIML2

[Pha00] describes the features of UIML2. They include:

- The implementation of user interfaces for any device without learning languages and APIs specific to the device.

[3] http://developer.android.com/guide/topics/ui/declaring-layout.html

- The natural separation between user interface code and application logic code.
- Simplification of internationalization and localization of UIs.
- It allows extension to support user interface technologies that are invented in the future.

Example

Code 2.3 shows the declaration of a simple UI that performs some action by calling external program logic. In detail, the <structure> element contains the actual UI definition, which, in this example consists of three parts: an "Input" part where the user can enter text, an "Output" part for the application to display results and an "Action" part where a call to the application logic can be made.

The <behavior> and <rule> tags encapsulate events that can happen on the UI, such as a button click. In the example, a click on the "Action" button causes the "actionPerformed" event to be executed.

Finally, the <peers> tag encapsulates two elements and the <presentation> element, which is not included in this example, can be used to hardcode additional information for specific UI toolkits (e.g. for the Java graphical user interface Toolkit Swing). The second and more important element is the <logic> tag. It is used to specify calls to and returns from the application logic. In this example, this causes a call to the method process of MyApp with "Input" as parameter. The result of the operation is then stored in the "Result" part. [Pha00]

Code 2.3 UIML2 example based on [Pha00]

```
<interface>
    <structure>
        <part class="Frame">
            <part name="Input" class="TextField"/>
            <part name="Result" class="Label"/>
            <part name="Action" class="Button"/>
        </part>
    </structure>
    <behavior>
        <rule>
            <condition>
                <event class="actionPerformed" part-name="Action"/>
            </condition>
            <action>
                <property name="text" part-name="Result">
                    <call name="MyApp.Process">
                        <param name="Value">
                            <property name="text" part-name="Input"/>
                        </param>
                    </call></property>
            </action>
        </rule>
    </behavior>
</interface>
<peers name="Java_Local">
```

```
26      <logic>
27              <d-component name="MyApp" maps-to="org.uiml.apps.MyApp">
28                  <d-method name="Process" maps-to="process" return-type="
                        string">
29                  <d-param name="Value" type="string"/>
30                  </d-method>
31              </d-component>
32      </logic>
33  </peers>
```

2.3.3 Runtime User Interface Description Language

The main idea behind [LC01] is to create a runtime UI description language for embedded devices in order to provide a unified user interface for a multitude of devices without the need of implement a completely new UI for every new device. It shall then be posible to display the UI on devices such as smartphones, other handheld devices or PCs.

The features of the proposed system include the UI description via the specified language, downloading the description from the described device and downloading and adapting the description according to the users preferences. This means, customized versions for different types of users should be available, such as end users, maintenance staff and manufacturers.

Technological Prerequisites

The authors conducted a search for technologies supporting their features and a set of more specific requirements, including:

- Platform independence: The UI description must be independent from the system it will be used on.
- Consistency: The notation has to be consistent across different environments.
- Unconventional I/O: Embedded devices have a much greater variety of inputs than usual desktop PCs with mouse/keyboard.
- Rapid prototyping: This allows users to be involved in the development process.
- Constraints: As embedded systems are facing constraints, they have to be specifiable by the description.
- Extensibility: The description has to be extendible.
- Reusability: Upon new products being developed, old designs shall be reusable.

The authors propose the use of XML since it meets all the requirements. Additionally, XML offers many additional features, such as XSLT conversion in order to create HTML/CSS code.

Furthermore, the authors discuss how the structure of the UI can be represented in XML. As XML is a tree structure, using this very structure for the user interface seems

obvious, and it is viable. As such, the main window of the user interface is the root. It contains building blocks consisting of user interface elements like buttons and sliders. The main window may as well include further sub windows. Interaction with the UI shall be described using abstract interaction objects. The abstract interaction objects represent a platform independent, abstract user interface widget (e.g. a button). The concrete interaction object is an implementation of an abstract one, depending on the platform (e.g. java.awt.Button for a Java graphical button). The concrete object has to be chosen according to the environment on which the description is being displayed.

Runtime User Interface Conversion

The system allows to transform a Java AWT based (embedded system) GUI into XML representation at runtime. This allows the creation of specialized GUIs on request. Moreover, this approach permits for any other objects to be serialized at their current state.

As an example, Code 2.4 partially shows the code of a described user interface. It demonstrates the basic tree structure of the main Window containing a widget in form of properties, whereas the widget itself is defined by properties as well.

Code 2.4 Example showing parts of the XML code for a described user interface based on [LC01]

```
<Application NAME="test.testUI"> [...]
    <Property NAME="name">main-window</Property>
    <Properties>
        <Property NAME="copyButton">
      <AIO CLASS="Button">
        <Properties>
          <Property NAME="name">button0</Property>
          <Property NAME="actionCommand">Copy</Property>
          <Property NAME="label">Copy</Property>
        </Properties>
      </AIO>
    </Property>
    [...]
  </Properties>
</Application>
```

Transfering the User Interface

After the user interface has been serialized into an XML description, it can be moved to another device. This new platform might use a different UI toolkit, as there are only abstract interaction objects in the description. The deserializer, however, needs to have the concrete interaction objects for the platform. The structure of the user interface will stay the same since it is defined by the XML, although it might not look exactly the same. After the UI has been rebuilt, the interaction with the embedded system works via remote procedure calls or similar technologies.

2.3.4 Remote UI for Embedded Systems

The work done by [GR11] introduces a concept of building a graphical user interface for real-time embedded systems in digital motor control test benches. Although this is a very specific application area, the system does, as the author specifically points out, have strong general purpose reusability.

The idea is to have a GUI that is able to directly identify, read and write all variables available within the embedded system. This is possible by using the map file of the compilation process for the embedded system control software. It contains the physical addresses of necessary elements – variables, pointers and arrays.

The following concept, consisting of four parts, will explain how this is achieved. The first part is the GUI software which runs e.g. on a PC, and allows the user to view and influence data from the embedded system. The second part is the communication interface which has to be supported by the embedded system, but could for example be USB or RS232. The third part is the protocol which is used to interchange information. The last part is the protocol processing function of the embedded system which reads and writes memory according to the protocol.

In order to achieve a meaningful structure within the information to be displayed by the GUI, the map file is converted into an XML file. Next to the automatically available data like memory address and data types, it is possible for the user to add additional data, e.g. variable grouping, colors, etc. to the XML file. This allows for much more flexibility and configurability than using only the map file directly.

2.4 Description Languages in Building and Home Automation

Nowadays, a great amount of different commercial standards for building and home automation are in existence. The one thing they all have in common is they need to make use of a number of small devices, sensors and actors. Therefore, they must to somehow map functionality to certain devices, which is usually done by a description language. They differ in communication technologies and protocols, but also in scale and security capabilities. While home automation focuses on automation and remote controlling of lightning and other electronic devices found in residential buildings, building automation systems go beyond that feature-wise, including automation of climate control, fire suppression, security systems and more. For example, BACnet[4] and LonWorks[5] are two very well known building automation standards. In home automation environments, standards like INSTEON[6] and Z-Wave[7], amongst others are deployed. Some standards are even open source, but an issue remaining for end users is the lack of interoperability between these technologies which is usually not given.

[4] http://www.bacnet.org/
[5] http://www.echelon.com/technology/lonworks/
[6] http://www.insteon.com
[7] http://www.z-wave.com

2.4.1 Z-Wave

Z-Wave is a relatively new commercial wireless communication technology for home automation. Z-Wave defines its own communication standard, on the physical as well as on the network layer, and its application layer.

Device Description in Z-Wave

Z-Wave device specification is based on so called command and device classes. They each represent certain device functionality. "basic command classes" need to be supported by all devices and represent functionality like setting values. Furthermore there are three levels of "device classes", the basic, generic and the specific device classes. They allow more for specific functionality supported by a device to be defined. The basic device class distinguishes between different routings. The generic device class defines the basic functionality a device offers. The specific device class allows a further, more fine-grained definition.[Pae11]

Device Description Example

A Schuko power adapter plug serves as an example. It's basic device class shall be "routing-slave" which allows it to be used at different locations. Its generic device class shall be a binary switch, as it represents on/off functionality. Its specific device class shall be a "binary power switch" which requires the device to support the functionality to switch of together with other devices within the Z-Wave network.

The description of a device, including its basic, generic and specific Class, it's supported Command Classes as well as other information is given in an XML[8] file. A List of Z-Wave device descriptions can be found in the Z-Wave device library [9].

2.4.2 Open Home Automation System

The "Open Home Automation System" (OHAS) [Tor08] is a technology supporting the integration of devices from different vendors into an open home automation network. Openness and vendor independence are the primary goals, as authors hope to support innovation in the market by this. The paper introduces two steps to achieve these goals. First, a scalable network allowing any number of devices and sub networks to be connected and accessible. Secondly, all devices have to be described by a common description language comprised of two parts: an interface description and a semantic description.

Device Description in the OHAS Environment

OHAS proposes a new and open XML-based description language. Any device described in this language could then enter the OHAS network and provide its functionality to participants within. XML has been used due to its several advantages and especially due to its ease of use as it is easily parseable as well as humanly readable.

Code 2.5 shows how OHAS uses formal language statements to interface devices, combined with natural language marking them up. The description of each service is, as

[8] http://www.w3.org/TR/xml/
[9] http://www.pepper1.net/zwavedb/

mentioned before, split into two parts: a semantic description and a syntactical description of how they can be accessed.

The example shows an OHAS device description example holding all necessary information for the system to address it (UID, IP), use it (input description), integrate it into a model (service description) and, if necessary, present it to a user, as it contains all the necessary information in humanly readable textual form (e.g. prettyName).

Code 2.5 Example of a lamp integrated into the OHAS based on [Tor08]

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <root>
3      <device description="Lamp E540"
4          prettyName="Power saving lamp"
5          UID="00:12:22:EE:11"
6          [...]
7          IP-address="192.168.1.151" port="2022">
8          <service description="Set new state"
9              name="SET_STATE" [...] >
10             <input description="input1"
11                 [...] >
12                 <possible value="1" H_meaning="ON" />
13                 <possible value="0" H_meaning="OFF" />
14             </input>
15         </service>
16     </device>
17 </root>
```

2.5 Description Languages for Sensor Networks

Sensor networks play an important role in future home automation systems. Single sensors are able to provide functionality such as reading RFID tags, sensing motion, temperatures, etc. while being cheap, small, mobile and even passively powered. A room equipped with several such sensors could automatically monitor all possible parameters and act upon them (e.g. turn on lights). Home automation is, however, by far not the only sector sensor networks are used in.

A multitude of different standards for describing such networks and ensure their dynamic extendibility already exist. [CH08] gives an overview of some of the most popular standards and discusses their advantages and disadvantages. The following sections will deal with some of these languages in more detail.

2.5.1 IEEE1451

The IEEE[10] 1451 standard, also "Standard for a Smart Transducer Interface for Sensors and Actuators" describes an open communication interface for connecting transducers to microprocessors and, furthermore, to networks. There is a family of standards related to IEEE1451. IEEE1451.0 contains a declaration of functions, commands and the transducer

[10] ttp://www.ieee.org/

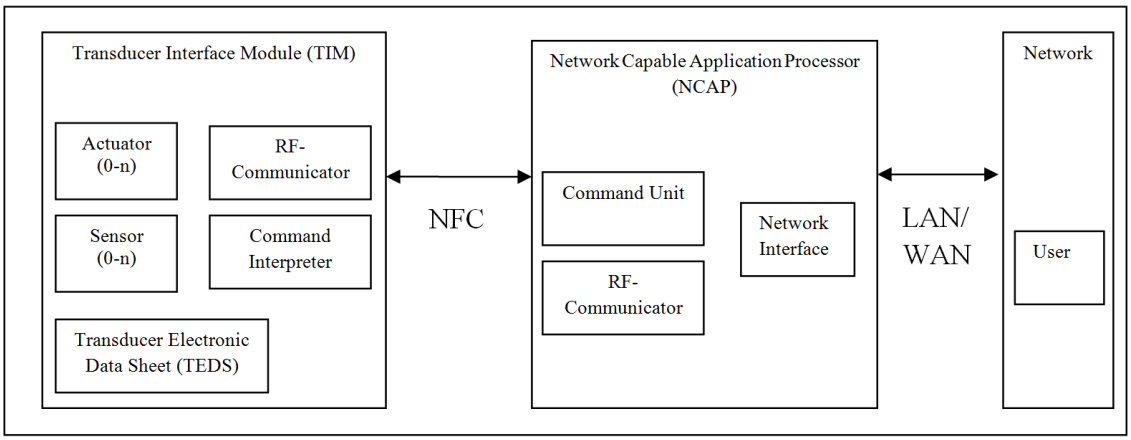

Figure 2.2: Overview of an IEEE1451.7 system based on [IEE10]

electronic data sheet (TEDS) building the basis for the following standards (IEEE1451.2 – IEEE1451.7). [IEE07] Amongst these there is also a standard for RFID communication, the IEEE1451.7 standard [IEE10].

In short, the IEEE1451 standards family defines an architecture that allows to interface a number of transducers via different communication technologies. An overview of the technology is given by 2.2 depicting a IEEE1451 based system that permits interaction between a user and a number of transducers.

Overview

TIM

The "transducer interface module" (TIM) consists of the TEDS for the sensors and actuators it contains as well as the logic to address them. It also serves as an interface for the NCAP and as such to higher level devices to use its functionality. Furthermore, the TIM is able to convert sensor values if necessary (e.g. analog values). An exact specification can be found in [IEE07].

TEDS

The "transducer electronic data sheet" (TEDS) is the core component of an IEEE1451 system. The TEDS is a standardized format used to store transducer data in a well defined way in order to allow and simplify access to the transducer across the network. It is usually attached to the TIM and it also serves to identify the connected transducers. Moreover, the TEDS contains the actual binary transducer functionality description. There are different kinds of TEDS, the most important ones being:

- Meta-TEDS: Allows access to all transducers connected to the TIM and contains common information.
- Transducer-channel TEDS: Describes measurement units and other information for a transducer.

- User's transducer name TEDS: Defines the name by which the transducer shall be identified.

There are several more, non-required, TEDS providing additional information for a system, if necessary. An exact specification can be found in [IEE07].

NCAP

The "network capable application processor" (NCAP) is a part of the system allowing for it to be connected to an outside network. Therefore, it is located between the TIM and the high-level user application. It is equipped with the logic to communicate with all connected TIMs and, moreover, has the ability to do data conversions and calculations based on the transducers data if defined to do so. An exact specification can be found in [IEE07].

2.5.2 Device Description Language

The "Device Description Language" (DDL) [11] is an XML-based markup language developed to describe the integration of sensors, actuators and devices (e.g. blood pressure monitor) and integrate them into an intelligent environment. [MPCL08]

The idea behind DDL is to simplify the process of interfacing devices for programmers as devices from different manufacturers usually have different protocols, even though their functionality might almost be identical. DDL aims to solve this problem by using an XML-based description of device interfaces, which leads to platform independence as well as human readability.

DDL Files

DDL aims to integrate described devices into the "Atlas software environment". This middleware system can keep track of connected sensors and address them using the device drivers stored in the middleware [HiYKR07]. DDL files are created via a web interface, once for each device. After creation, the server checks its validity and, should it be valid, the DDL Parser and the Device Bundle generator create a bundle project for the device. This bundle is then uploaded to the device repository within the Atlas middleware system from where it can be accessed and easily used. This file contains all the information necessary to access the full functionality of a device.

DDL Structure and Example

As for the structure of the DDL XML file, a shortened and simplified example is shown by Code 2.6. DDL breaks devices down into three categories: sensors (provide only data to a user), actuators (which take only data from a user) and complex devices (do both). DDL allows a variety of descriptions for the device beyond its actual functionality, such as a text based description for its functionality as well as a physical description (height, length, operating environment, etc.).

[11] http://www.icta.ufl.edu/atlas/ddl/

Code 2.6 DDL XML example based on [MPCL08]

```
1  <DDL version="1.3">
2     <Device>
3         <Description>
4             <Name>AnDBPMonitor</Name>
5             [...]
6         </Description>
7     <Interface>
8          <Signal id="s1">
9              <Operation>Input</Operation>
10             <Type>Protocol</Type>
11             <Measurement>Digital</Measurement>
12             <Unit />
13             [...]
14         </Signal>
15         <Reading id="r1">
16             <Type>Physical</Type>
17             <Measurement>SysDiaDiff</Measurement>
18             <Unit>mmHg</Unit>
19             <Computation>
20                 <Type>Formula</Type>
21                 <Expression>r1 = s1.substring(2,4);</Expression>
22                 [...]
23             </Computation>
24         </Reading>
25         [...]
26     </Interface>
27     </Device>
28 </DDL>
```

Code 2.6 illustrates how DDL describes the functionality of a device. Every device can contain multiple so-called "signals". A signal is defined to be some measurement that can either serve as input to or output from the device. Its description contains meta-information, like a unit and ranges. Based on the signals, "readings" can be defined. They take a signals value as input and do computations based on them. These computations are described within the DDL code. In the code example, the computation does not perform any calculations, but only takes some bytes from the sensor reading.

2.6 XML as a Description Language

XML is used as the basis for many user interface description languages as well as for description languages describing hardware devices. For example, in home automation and sensor network environments. This is as well reflected by the previous chapters.

This chapter deals with the possibility of using the XML Markup language explicitly to describe and interact with embedded systems. Furthermore, should XML proof viable for this task, some basic design rules shall be looked at, helping to establish a clean XML design for the practical part of this thesis.

2.6.1 XML for Embedded Systems

The article [Jon05] provides an example of how embedded systems can benefit greatly by XML usage for communication and data exchange. This specific example features medical devices integrating themselves into a network infrastructure. An embedded system manages the available medical devices describing themselves via a name, address, location and whether they are in use. In order for the devices to generate real-time data, a DOM XML library was used to allow simple updates of the XML structure, however without recreating the document as a whole. One of the decisive advantages listed by the article is the so-called XSLT transformation, which enables devices to be compatible across data formats, e.g. by the potential to transform the standard XML document into .csv or a similar format.

The usage of XML is very common in data processing, structured communication protocols and also service descriptions. A great amount of XML-based languages is already in existence for a multitude of different fields of application.

The thesis [Huf03] investigates the role of XML within embedded systems and especially collaborative embedded systems.

The author states a number of risks and opportunities arising from the usage of XML. In detail, these are:

- Limited resources: A very common and well known restriction of embedded systems. This can limit the possibilities of deploying an XML parser on the embedded system and restrict its functionality to very simple XML structures.
- Expenses: XML can potentially help simplify development of new systems and reduce interoperability cost with future systems. As a consequence, development would be faster and overall costs could be reduced.
- Separation of structure: XML allows the separation of the actual data, syntactical description of the structure and the presentation of the data within the structure. This permits the system to automatically create XML structures, for example, to provide a client with information stored on the system.

The author draws several conclusions, most of which suggest XML is, in general, very well suited for embedded systems.

On the one hand, using XML as a protocol proofs to be very flexible, however, the specification produces a lot of unnecessary overhead in communication between devices. On the other hand, the great advantage of using XML as an open standard in a communication protocol is its of compatibility.

Furthermore, applying XML as the language for describing an embedded systems functionality is viable. As XML is a very good standard to describe data structures, it is as well suited for describing devices. The data describing a device can be stored in a structured way. The definition of the description can be embedded into the device as well.

2.6.2 XML Design Guidelines

As XML turns out to be the most viable candidate to be used to describe the devices attached to the NFCI system, further research shall be conducted into potential problems arising from XML usage in order to avoid them. A closer look shall be taken at best practices for designing XML specifications. This shall ensure best possible readability and understandability as well as good extensibility without requiring changes to the existing structure later on.

The author of [Ogb04] writes about some of the most common issues when designing XML specifications, especially, when to use elements versus when to use attributes. The author gives some essential concepts that should help programmers when developing an XML specification. He explicitly warns of using exclusively elements and no attributes at all, which causes an essential feature of XML to be completely left out and decreases readability.

The "**Principle of core content**" says that information essential to the meaning of the XML's content should always be in elements. This goes as well for data to be read by humans as data read by machines. Any data that is peripheral and/or just included in order to help machines process data, should be in elements. This guideline helps keep essential content from being cluttered by subsidiary information. In a more technical language, this could be expressed as "data goes into elements, metadata into attributes". A negative example given by the author for this principle is a document title being placed in an attribute.

The "**Principle of structured information**" specifies that structured information in general and especially information that might be extended later on should always go into elements. Information should go into an attribute if it belongs exclusively to an element and its structure is not subject to change. A "date" serves as an example to this principle. As it is a single piece of information with a fixed structure it belongs in an attribute. As a second example, the author discusses how to handle a person's name. It is usually subject to extensions (e.g. honorifics, further names), thus should usually neither be an attribute nor a single element.

The "**Principle of readability**" recommends to always put data to be read by humans (directly out of the XML structure) into element tags.

The "**Principle of element/attribute binding**" specifies to put information that should be modified by an attribute into an element. An attribute defines some part of the element more specifically. In general, attributes should not modify attributes.

Chapter 3

Design

This chapter provides an overview of the new NFC Interface (NFCI) system. The hardware will not undergo any changes and is therefore the same as described in chapter 2.2.2. All software components belonging to the system will, however, be changed and extended. These components are part of the system and will incorporate changes:

- Android smartphone app
- NFC Interface firmware
- Target Device functionality description
- Target Device software (PC applications for demonstration purposes, representing the Target Devices behavior and appearance including its nodes.)

This chapter is divided into several sections starting with use cases and the system requirements the design is built upon. It also includes the NFCI system architecture, an overview of the required Android functionality and the concrete design of all software components being developed as a part of this thesis.

3.1 Use Cases

This section shows two use cases used to refine the requirements for the NFCI system. Moreover, they shall give a basic overview of the kind of functionality that can be expected of the NFCI system.

3.1.1 Microwave Oven

Figure 3.1 depicts the microwave oven use case, showing how a user can interact with the microwave via his Android smartphone. First, the user wants to get an overview of his devices capabilities via the app. He then wants to be able to have the microwave cook using adjustable time and power settings. This feature shall be extended by providing usage suggestions for the microwave to the user. Also, the user wants to configure his microwave in order to automatically start cooking at a certain time. At last he also wants to be informed about the current status of the microwave, in this case seeing how much

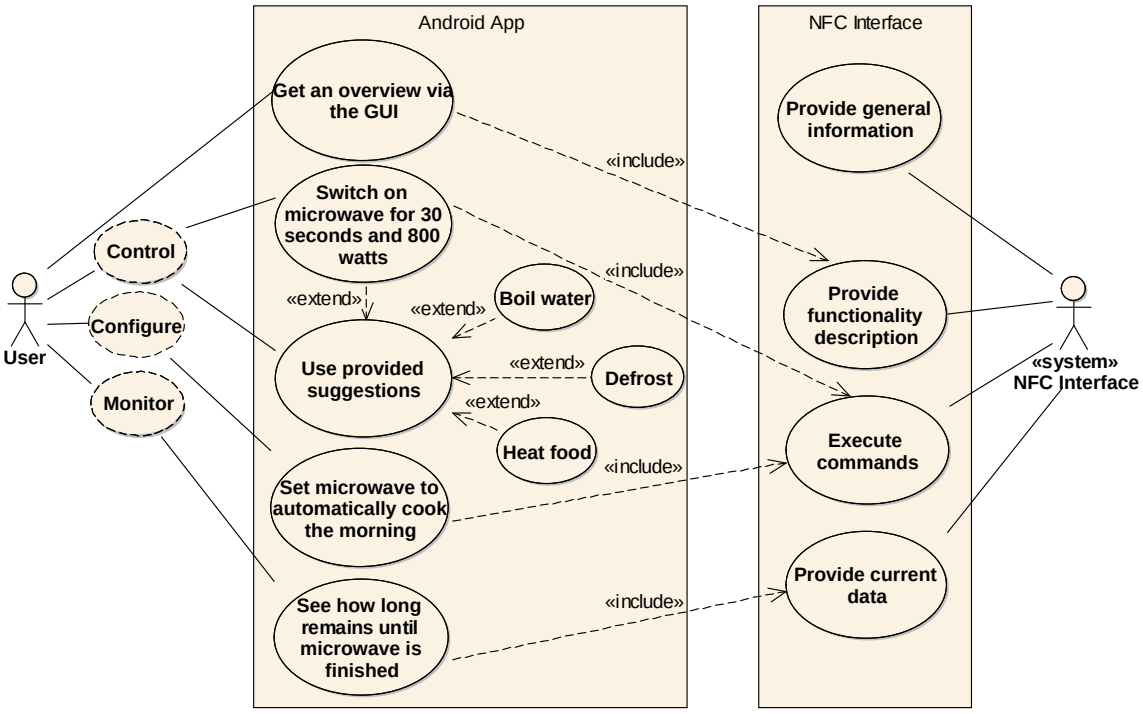

Figure 3.1: Use case for the microwave oven

time remains until the microwave has finished cooking. The NFCI provides access to all of the functionality to the user.

3.1.2 Energy Meter

Figure 3.2 depicts the energy meter use case, showing how a user and a vendor can interact with the Energy Meter via their Android smartphones. The user wants to be able to see as well the current power consumption of his running consumer devices as the total power consumed by them. These consumer devices could for example be a TV, a radio or lights. The vendor wants to be able to reset the data collected by the energy meter, however this feature has to be restricted to the vendor himself. Therefore he needs to be able to authenticate himself as such to the NFCI. The NFCI provides access to all of the functionality to the user and the vendor.

3.2 Requirements

The following tables (3.1 - 3.6) contain functional and non-functional requirements for all components of the NFCI system. These requirements are based on the use cases as well as the NFCI requirements given in [Bas13]. For some of the requirements, a set of actors having different rights to access some of the Target Devices functionality play an important role. The following actors have been defined:

- The user: A person who owns an Android NFC enabled Android smartphone with the NFCI app installed and does aswell own an NFCI compatible device.

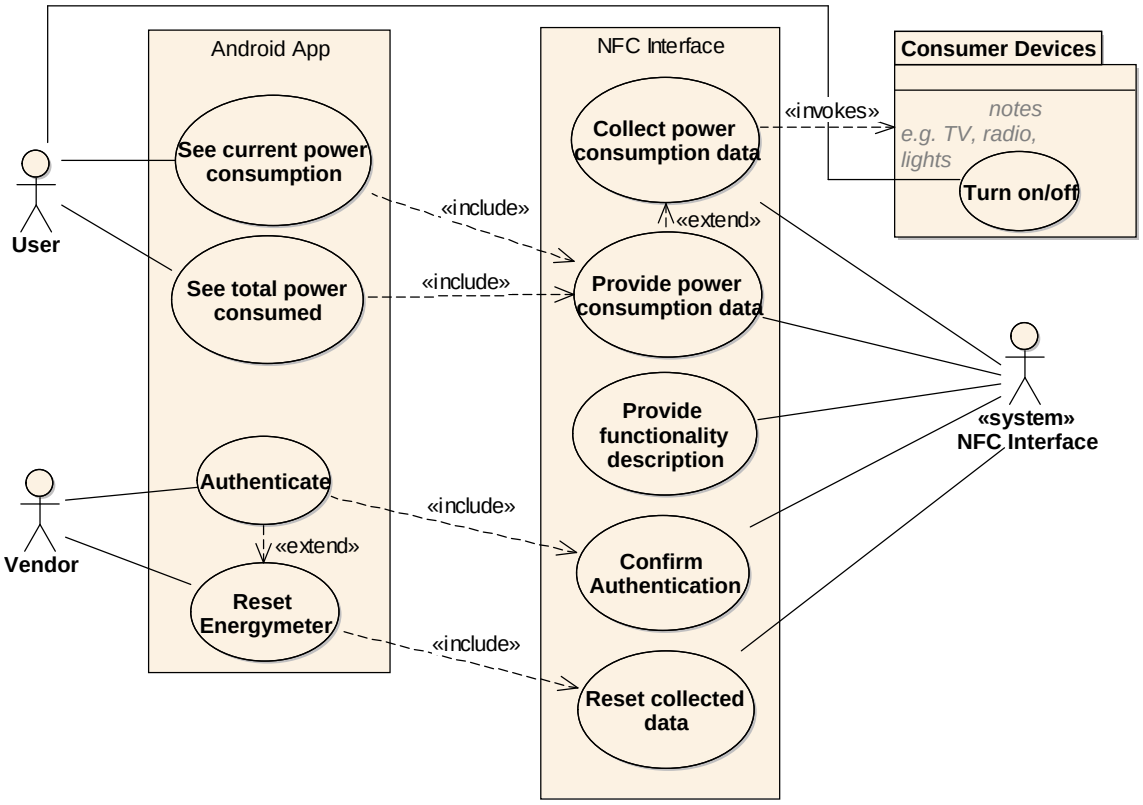

Figure 3.2: Use case for the energy meter

- The service technician: A technician who needs to be able to maintain the device connected to the NFCI and access functionality restricted to customers.

General system requirements	
Number	Requirement
1	The user shall be able to interact with the NFCI, directly from the NFC-enabled Android smartphone.
2	The user shall be able to interact with attached NFCI compatible nodes, directly from the NFC-enabled Android smartphone.
3	The user shall be able to monitor the internal states of the node.
4	The user shall be able to configure the node, according to provided internal states and configuration parameters.
5	The user shall be able to control the node.
6	It shall be possible to access and list all possible internal states of the node.
7	It shall be possible to upload data to be stored on the Target Device.
8	The NFCI shall support an activation process. Only an activated NFCI shall provide access to the Target Device nodes.
9	The communication with the NFCI shall work via the NFC Forum Tag-Type4 protocol.

Table 3.1: General system requirements

Target Device Description requirements	
Number	Requirement
10	A description for the Target Device shall be introduced to allow the exact definition of its functionality.
10a	The description shall be able to describe any node functionality.
10b	The description shall be able to describe NFCI functionality, such as the possibility to upload new data into its storage.
10c	The description shall feature the possibility to provide usage suggestion to the user.
10d	The description shall contain any information necessary to allow clear identification of its content by the user.
11	The description must support the possibility to be parsed automatically without any actor involvement.
12	The description must be extensible.
13	The description must be small enough to be stored on the Target Device's non-volatile memory.

Table 3.2: Target Device Description requirements

General Android app requirements	
Number	Requirement
14	The Android app shall support interaction with the Target Device and allow access to its features.
15	The app shall automatically detect NFCI compatible NFC tags.
16	The user shall get feedback when he touches a NFCI compatible device with his smartphone.
17	The app shall be able to retrieve the Target Device Description.
18	The app shall remember Target Devices the user has connected to before.
19	The app shall store the Target Device Description for future use.
20	The app shall provide status dialogs for large data transmissions (e.g. Target Device description).
21	The app shall support the Target Device activation process by allowing activation as well as deactivation of the Target Device.
22	The app shall support the creation of a dynamic graphical user interface according to the functionality given by the Target Device Description.
23	The graphical user interface shall allow the user to interact with all Target Device functionality.
24	The user shall be able to see the current configuration parameters and monitor values of each node.

Table 3.3: General Android app requirements

Android app user interface requirements	
Number	Requirement
25	The GUI shall structure the information contained in the Target Device Description clearly.
26	It shall be possible for the user to distinguish and navigate between the nodes via names and appearance.
27	It shall be possible for the user to clearly and simply identify elements contained within the GUI.

Table 3.4: Android app user interface requirements

Target Device requirements	
Number	Requirement
28	A general information NDEF message shall be available to any NFC-enabled device informing a user about the Target Device.
29	The description must be available to any Android NFC-enabled device with the NFCI app installed as an NDEF message.
30	The Target Device shall be able to retrieve data from the NFCI app and provide it as an NDEF message later on.
31	All communication involving NDEF messages shall work via the Tag-Type4 protocol.

Table 3.5: Target Device requirements

Actor specific and security requirements	
Number	Requirement
32	All actors shall be able to read the general device information of the Target Device.
33	All actors shall be able to monitor the data of the Target Device.
34	The service technician shall be able to upload a new firmware to the smart meter.
35	The Android app GUI shall represent the restriction of certain functionality according to access levels.
36	The Android app GUI shall provide the possibility for the user to authenticate in order to gain access to the restricted functionality
37	The authentication process shall be done by the Target Device.

Table 3.6: Actor specific and security requirements

3.3 Architecture

The difference of the new architecture (depicted in Figure 3.3) compared to the architecture described in chapter 1.2 lies in the dynamic description of Target Device functionality by the Target Device Description (TDD), which is stored by the NFC capable controller and is retrievable and usable by the NFCI app. An exact overview of the changes is given in this chapter. The hardware will remain unchanged compared to the previous version of the system. For an exact description of the hardware see [Bas13].

NFC Interface App (1): The app will be remade from scratch. It communicates with the Target Device and is able to retrieve any information necessary to identify the Target Device, including the TDD. It can build a graphical user interface (GUI) representing any functionality described in the TDD and communicate with the Target Device in order to access any functionality supported by the Target Device.

NFC Module: The Android NFC-enabled smartphone is equipped with NFC hardware as well as an API for using its functionality. This component is used by the NFCI app.

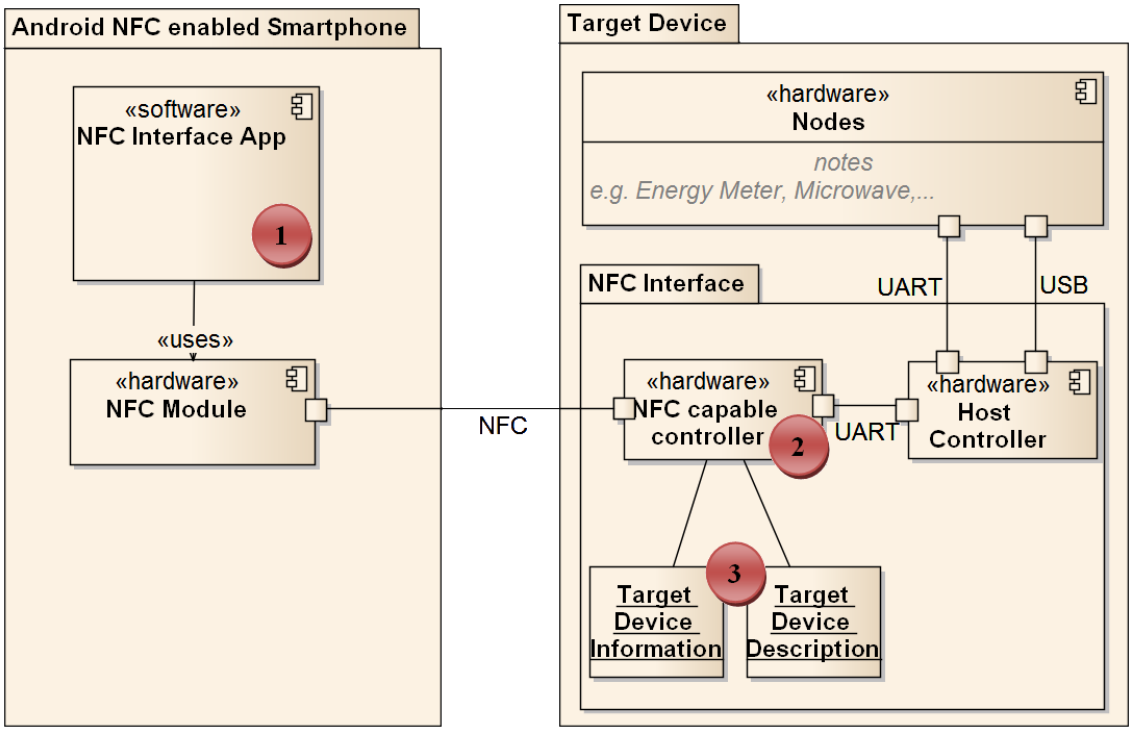

Figure 3.3: Architecture of the new NFC Interface system

NFC Interface: The NFC Interface is the hardware component that allows remote access to the Target Device's nodes which, in turn, contain the actual functionality a user would be interested in.

NFC capable controller (2): This component is a part of the NFCI and is equipped with the NFC hardware. Its firmware has to support the required communication protocols for the NFCI app to retrieve any stored information as well as communicate with the nodes.

Host Controller: It handles communication between the NFC-capable controller and the Target Device's nodes. In other words, it allows access to the nodes via the NFCI.

Nodes: Nodes contain the actual functionality a user would want to access. For example, when attaching the NFCI to a microwave oven, the microwave is a node.

Target Device Information and Description (3): These components identify the Target Device and its functionality. The Target Device Information is a set of data allowing the unique identification of the Target Device. The TDD contains a structured compilation of all functionality the Target Device has to offer to a user. Technically the description is an XML-based descriptive markup of the Target Device functionality.

3.4 The Android Platform

In order to understand the design of the NFCI app's functionality as well as its implementation and code, it is necessary to know the basics behind the Android development framework and the operating system itself.

3.4.1 Android Operating System

The Android OS has become the most widely used smartphone operating system in the world. A simple online search reveals Androids market share is at about 70% in Europe and 55% in the United States [1].

Android poses many challenges to developers, not only because of the great variety of smartphones and tablets in existence, featuring different hardware and screen sizes, but also for its variety of OS versions. For many reasons, including their hardware not being powerful enough, as well as vendor-specific operating system modifications and business strategies, many older Android smartphones will never receive updates to more recent versions of the operating system (4.0+). Still more than 30% of all Android devices are running versions below 4.0 [2].

For app developers this often means having to deal with the additional complexity of supporting different versions of the operating system. Older versions do not support some of the newer framework features, others do, however, get made available on older Android versions through the official Android Support Library [3]. It is especially important to decide which Android operating system is to be supported by the app.

As for the NFCI app, it can be argued that legacy support is not an important issue since it is only a concept and not even the NFCI itself is currently available to the outside world. Therefore, the app will be designed to work with versions above Android 4.0.3 (API 15).

3.4.2 Diversity of Devices

As mentioned before, Android is being used on a wide range of different devices. The SDK offers a flexible layout system supporting different presentations for screen sizes and density. Some of the most important concepts developers have to follow, in order to ensure the Android SDK can adapt the screen content to fit different devices, will be explained here.

Configuration qualifiers: Android can select a number of different resources depending on the current devices characteristics. Those characteristics are: screen size, screen density, screen orientation (landscape/portrait) and aspect ratio. This allows the dynamic usage of resources specifically designed to fit certain characteristics. An overview of all configuration qualifiers is given below. The most important question is: When does it make sense to use different layouts? For example, when deploying an app (designed for a normal size screen) to a small device it might happen that some views are overlapping and

[1] http://www.forbes.com/sites/chuckjones/2013/09/30/iphones-market-share-down-prior-to-5s-5c-launch-with-windows-almost-double-digits-in-europe/ Accessed: 23.10.2013

[2] http://developer.android.com/about/dashboards/index.html Accessed: 23.10.2013

[3] http://developer.android.com/tools/support-library/index.html

just don't fit on the screen. Here, a new layout should be introduced to address this issue. Next, when deploying an app designed for a normal size screen on a large screen device, for example a tablet, the UI components might look stretched. While this will not make the app function any worse, it might feel to the user like the app has not been designed well. Therefore, a new layout for large or xlarge screens should be introduced in this case as well. An overview of the configuration qualifiers is given in Table 3.7[4].

Screen characteristics	Qualifier	Description (Resource for...)
Size	small	small size screens (at least 426dp x 320dp)
	normal	normal size screens (at least 470dp x 320dp)
	large	large size screens (at least 640dp x 480dp)
	xlarge	extra large size screens (at least 960dp x 720dp)
Density	ldpi	low density screens (120dpi)
	mdpi	medium density screens (160dpi)
	hdpi	high density screens (240dpi)
	xhdpi	extra high density screens (320dpi)
	nodpi	not density specific, will not be scaled
	tvdpi	special density, intended for televisions (213dpi)
Orientation	land	screens in landscape orientation mode
	port	screens that are significantly taller in portrait mode, and wider in landscape mode
Aspect ratio	long	screens in landscape orientation mode
	notlong	screens that are similar to the standard configuration

Table 3.7: Android configuration qualifiers based on [4]

3.4.3 Android Programming Principles

This section shall give an overview of all Android technology required for the NFCI app. It is the basis for the implementation conducted later on. Reading this chapter is suggested if you are not familiar with the Android SDK.

Activities

In order for any UI to be shown by an Android app, one component is always required: An *Activity*. The Google Android developer documentation defines Activities as follows: *"An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface."* [5]

Furthermore, it is important to know that only one activity may be running in the foreground. Note that other activities may still be visible while in the background if the

[4] http://developer.android.com/guide/practices/screens_support.html
[5] http://developer.android.com/guide/components/activities.html

foreground activity is not a fullscreen activity. Whenever an activity gets started and another one has to be pushed into the background, the activity lifecycle allows them to take actions according to what state they are moving to. The official developer documentation defines this as follows:

"When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state - whether the system is creating it, stopping it, resuming it, or destroying it - and each callback provides you the opportunity to perform specific work that's appropriate to that state change." [5]

Some of the methods representing the activity lifecycle[6] have to be implemented in order for the app to achieve meaningful functionality. Note that the activity lifecycle is of major importance to any Android app! Some of the Activity class' most important methods will now be explained in more detail:

- *Activity.onCreate()* is called when an activity Is being created and has to set the content to be displayed by it. This method has to be implemented in any case.
- *Activity.onResume()* is called just before an activity is moved to the foreground.
- *Activity.onPause()* is called for the current activity when another activity wants to get into the foreground.

Fragment

Another component of high importance is the so-called *Fragment*[7]. Using fragments for a UI offers far more flexibility than just using activities, especially when adaptations for tablet are planned. The Google Android developer documentation defines fragments as follows:

"A Fragment represents a behavior or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities)" [7]

The *Fragment* also offers some specialized sub-classes. All their features could, of course, as well be implemented into a fragment derived from the base class. The most important Fragment sub-classes will now be explained:

- *ListFragment*: Contains a list of items being managed by an adapter, and provides default methods for managing the list content, such as listening to clicks.

[6] http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle
[7] http://developer.android.com/guide/components/fragments.html

- *PreferenceFragment*: Offers the functionality to display preference objects (XML-defined preference hierarchies) when building an app preference screen.
- *DialogFragment*: Allows displaying a (usually non fullscreen) dialog window floating on top of the current screen.

Combining Activities and Fragments

As described in chapter 2.3.1, Android UIs can be built in two ways, either by declaring UI components in an XML file, or by adding them programmatically at runtime. Fragments can also be added to an XML layout by using the <fragment> tag. Programmatically, a fragment can be added dynamically via a *FragmentTransaction*. Every Activity has a *FragmentManager* belonging to it. The *FragmentManager* is aware of any fragment that is part of its activity and can be used to commit fragment transactions.

View

The Google Android developer documentation defines a *View* as follows:

"This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.). The ViewGroup subclass is the base class for layouts, which are invisible containers that hold other Views (or other ViewGroups) and define their layout properties." [8]

It seems obvious the *View* class is of great importance as all UI components (so-called widgets) are derived from it and, therefore, no meaningful user interface can exist without a combination of view objects. An overview of the widget hierarchy is given in Figure 3.4. The most important widgets for the NFCI app will most likely include:

- *TextView*[9]: Displays text to the user; configured not to be editable.
- *EditText*[10]: Displays text to the user; configured to be editable.
- *Button*[11]: A push button, which can be pressed or clicked to perform an action.
- *CheckBox*[12]: A 2-state button, that can either be checked or unchecked.

[8] http://developer.android.com/reference/android/view/View.html
[9] http://developer.android.com/reference/android/widget/TextView.html
[10] http://developer.android.com/reference/android/widget/EditText.html
[11] http://developer.android.com/reference/android/widget/Button.html
[12] http://developer.android.com/reference/android/widget/CheckBox.html

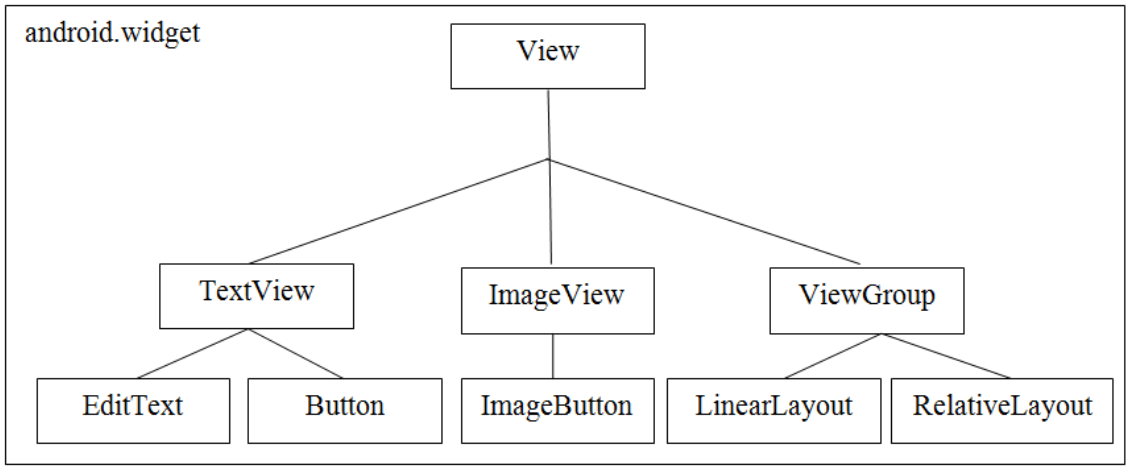

Figure 3.4: Simplified overview of the Android widget hierarchy

ViewGroup

View groups can be thought of as view containers allowing different methods of aligning their contained views. The Google Android developer documentation defines the *ViewGroup* class as follows:

"A ViewGroup is a special view that can contain other views (called children.) The view group is the base class for layouts and views containers. This class also defines the ViewGroup.LayoutParams class which serves as the base class for layouts parameters." [13]

Within the NFCI app, several different view group objects will be deployed for different purposes. The most important view groups for the app will most likely include:

- *RelativeLayout*[14]: Aligns its children according to layout parameters, allowing the definition of their position relatively to each other or their parent.
- *LinearLayout*[15]: Aligns its children in a single row or column.
- *GridLayout*[16]: Aligns its children into a rectangular grid.
- *Spinner*[17]: Displays a dropdown list of items for the user to choose from.

The *ViewGroup.LayoutParams*[18] class is of great importance in combination with the view groups mentioned above. It allows attaching layout information to each child of a view group. Note that they differ for different view groups (e.g. *RelativeLayout.LayoutParams* and *LinearLayout.LayoutParams*) as they vary in their layout process.

[13] http://developer.android.com/reference/android/view/ViewGroup.html
[14] http://developer.android.com/reference/android/widget/RelativeLayout.html
[15] http://developer.android.com/reference/android/widget/LinearLayout.html
[16] http://developer.android.com/reference/android/widget/GridLayout.html
[17] http://developer.android.com/reference/android/widget/Spinner.html
[18] http://developer.android.com/reference/android/view/ViewGroup.LayoutParams.html

AsyncTask

The *AsyncTask*[19] is an effective way to handle background computations and other time consuming operations that should not run on the UI thread, as this would cause unresponsiveness, while still being able to publish results back to the UI.

NDEF Record Types

NDEF records support the user of content types, such as MIME types or URIs in order to describe their content. This allows the Android system to dispatch found tags[20] to certain apps automatically. For example, a MIME type "text/xml" would show a list of all apps supporting this specific MIME type. NDEF messages may also contain a so-called "external type" which allows specifying a concrete package to be called on the Android system. This can be used to automatically launch the NFCI app upon touching a corresponding tag. Note, a full specification of the NDEF record types can be found at [For06b].

3.4.4 Android Design Principles and Guidelines

Every person operating a smartphone is accustomed to the specific way the UI looks, behaves and feels like. For instance, a list that contains more items than the screen can show at once has to be scrollable. Every single smartphone user knows how to scroll this list, namely by swiping with his fingers in the direction the list continues. Now, while scrolling the list, the user will look for a bar indicating how long the list actually is and once he has reached the end of the list he will expect some visual representation of this fact as well. Luckily all this behavior is so standardized in the Android framework, it works by default. However, think of an app that explicitly prohibits to scroll via swiping, but instead requires the user to click on a "scroll up" and "scroll down" button. This kind of design would go against what any user expects and no one would want to use it.

On modern smartphones a great deal of such, mostly already subconscious, usage paradigms exist. While it is not the main focus of this thesis to investigate and apply state of the art usability techniques, a good level of awareness for the topic is required for any developer aiming for an app to be accepted by its users. The Android developer documentation features a good introduction into the topic, including a list of what to do in order to ensure good usability [21] [22]. Some of the most important concepts will now be explained.

Gestures

Gestures are the primary way a user controls content on the smartphone screen. They play an essential role in every app, and will be thus be important for the NFCI app as well. The most relevant and important gestures based on [23] are:

[19] http://developer.android.com/reference/android/os/AsyncTask.html
[20] http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#tag-dispatch
[21] http://developer.android.com/design/index.html
[22] http://developer.android.com/design/get-started/principles.html
[23] http://developer.android.com/design/patterns/gestures.html

- Touch: Triggers the functionality assigned to a UI component by default. Can distinguish between press and lift up, e.g. a button changes its color on press, triggers its functionality on lift up.
- Long press: Triggers functionality differing from a simple touch, such as entering a data selection mode.
- Swipe: Navigation between views in a hierarchy and scrolling overflow content.
- Drag: Rearrange data, e.g. move something from one location to another.

Action Bar

The Action Bar [24] usually is a persistent piece of the UI on top of every screen throughout the app. The Action Bar includes several elements that help in giving the user intuitive control over an app. Using an Action Bar can go a long way in making the app feel similar to other apps designed for the Android system. Some advantages of using an Action Bar are:

- It makes important actions prominent and accessible in a predictable way (such as "New" or "Search").
- It supports consistent navigation and view switching within apps.
- It reduces clutter by providing an action overflow for rarely used actions.
- It provides a dedicated space for giving an app a unique identity.

Additionally, Figure 3.5 now illustrates the basic functionality of an Action Bar.

1. App icon: Can be seen as the apps personal image; may contain an up-caret allowing backwards navigation when not currently on the top level screen.
2. View control: Allows a user to switch between Views.
3. ActionButtons: Show the most important actions for the app.
4. ActionOverflow: Items that do not have enough space to be displayed amongst the ActionButtons are moved here.

The NFCI app will also feature an Action Bar because of the many advantages mentioned above.

[24] http://developer.android.com/design/patterns/actionbar.html
[25] Source: http://developer.android.com/design/media/action_bar_basics.png

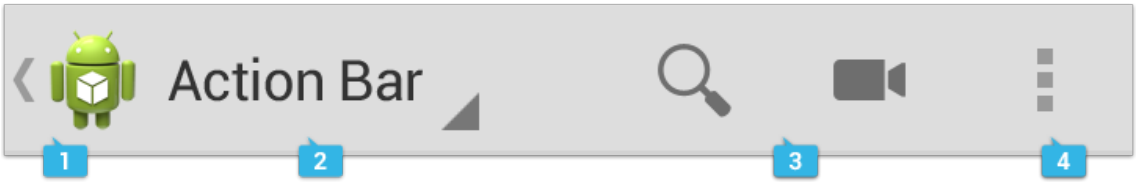

Figure 3.5: An example for the Action Bar on the Android OS[25]

Navigation

Android provides different ways to navigate through the content of an app. It is necessary for the app implemented as a part of this thesis to navigate through dynamically created content. In data-driven apps, such as the Google Play Store[26] app, depicted in Figure 3.6, the navigation structure should allow and help the user to view and manage the data he is looking for. Categories permit data to stay in the users focus instead of having to deal with hierarchies. In Figure 3.6, the categories are closely related and are not shown on the screen as a whole at any one time. In Figure 3.7, depicting the Youtube[27] app, the categories are relatively unrelated and, therefore, fixed tabs have been used where all categories are visible at the same time in order to ensure the user is able to keep an overview of the functionality.

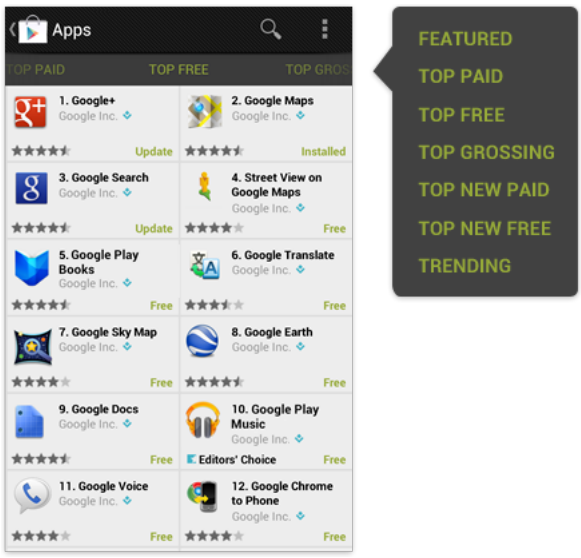

Figure 3.6: Scroll tab navigation [28]

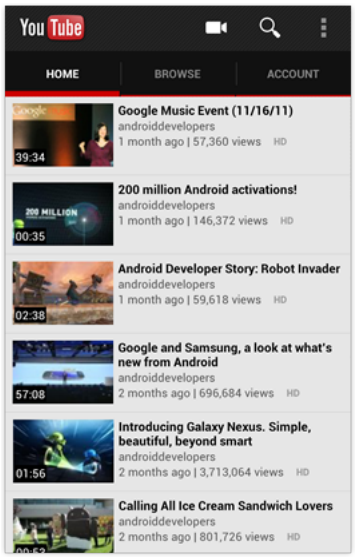

Figure 3.7: Fixed tab navigation [29]

[26] https://play.google.com/
[27] http://www.youtube.com/
[28] Source: http://developer.android.com/design/media/app_structure_scrolltabs.png
[29] Source: http://developer.android.com/design/media/app_structure_fixedtabs.png

3.5 Design – NFC Interface App

3.5.1 Mockups

The user interface mockups depicted in Figure 3.8 provide a first impression of the Android app UI. They only serve as a prototype for designing, planning and demonstrating functionality as well as obtaining feedback based on them. They are just pictures, no code involved, and created using the program Pencil[30]. On the left side of Figure 3.8, the app home screen is shown containing a list of all devices the app has previously been connected to. On the right side of Figure 3.8, the UI for a microwave is presented. It includes the functionality to start and stop the microwave, set it to cook automatically at a certain time and also to monitor its power consumption and time remaining.

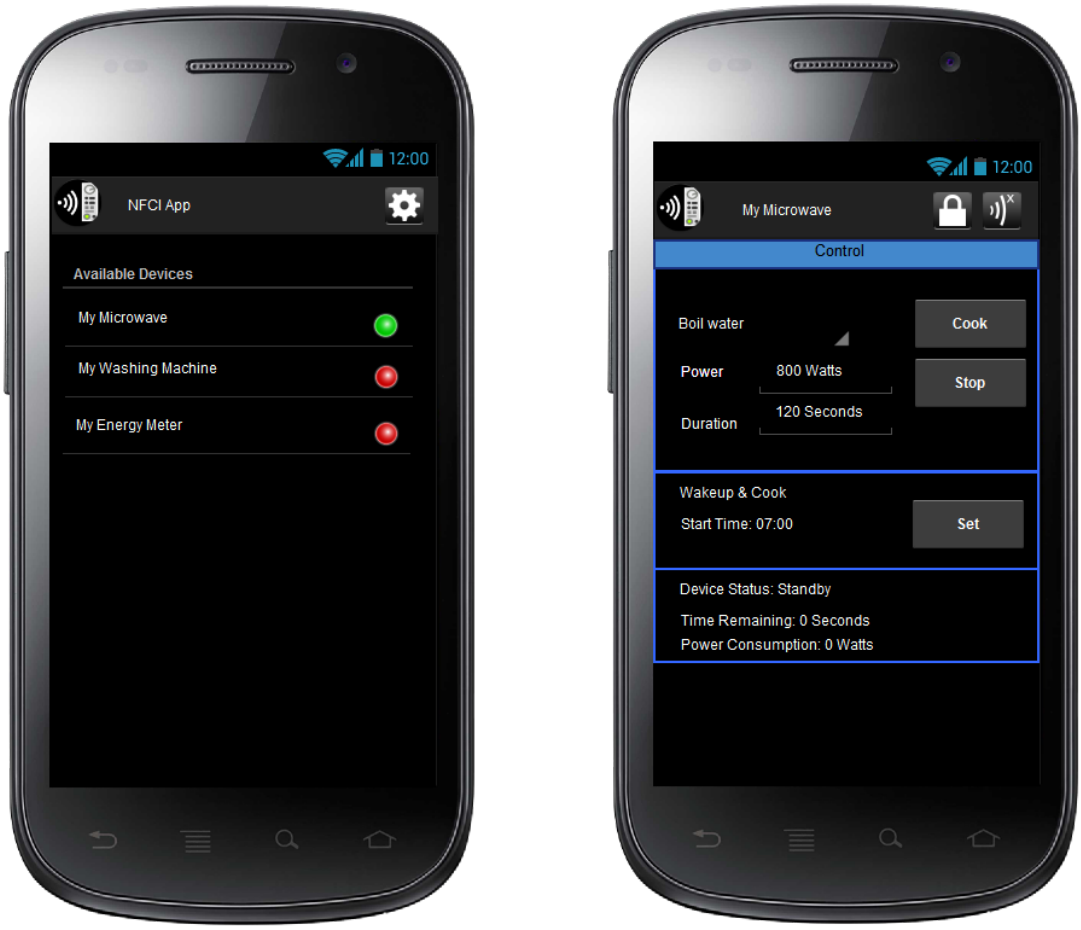

Figure 3.8: Mockups of the NFC Interface app

[30] http://pencil.evolus.vn/

3.5.2 Functionality

A basic overview of the app's functionality including its collaboration with the user is depicted by the activity diagram in Figure 3.9. The diagram shows, in a highly abstracted way, what the user can do at which point and what kind of action is taken by the system upon a certain user action.

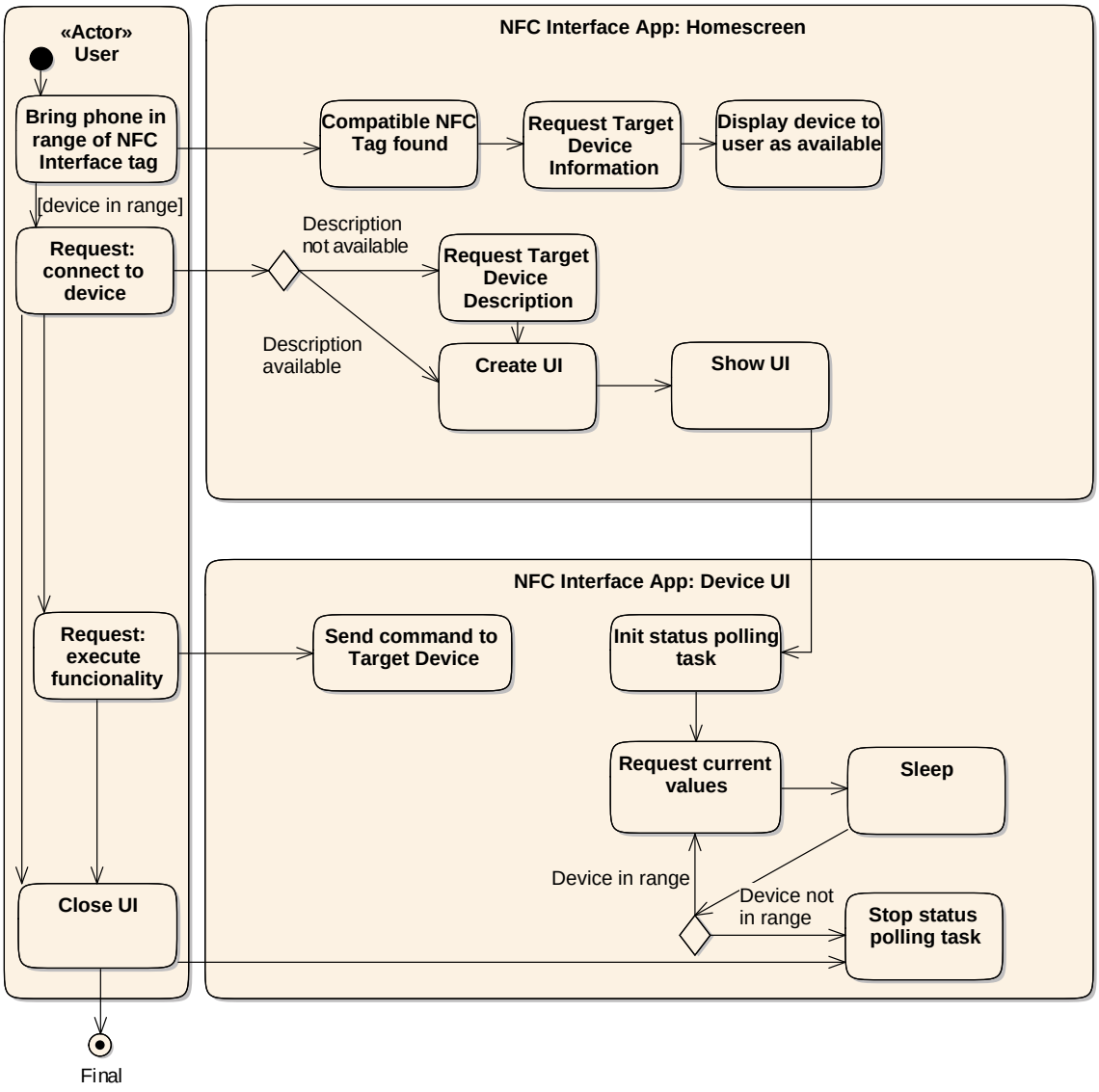

Figure 3.9: NFC Interface app activity diagram

As the app functionality has to be quite complex, its design will be presented divided into its packages. Every package has one or more unique tasks to fulfil within the app.

User Interface Components

The "app" package, depicted in Figure 3.10, contains all functionalities related to providing a UI to the user, as well as interacting with the system in general. The UI creation is not part of this package.

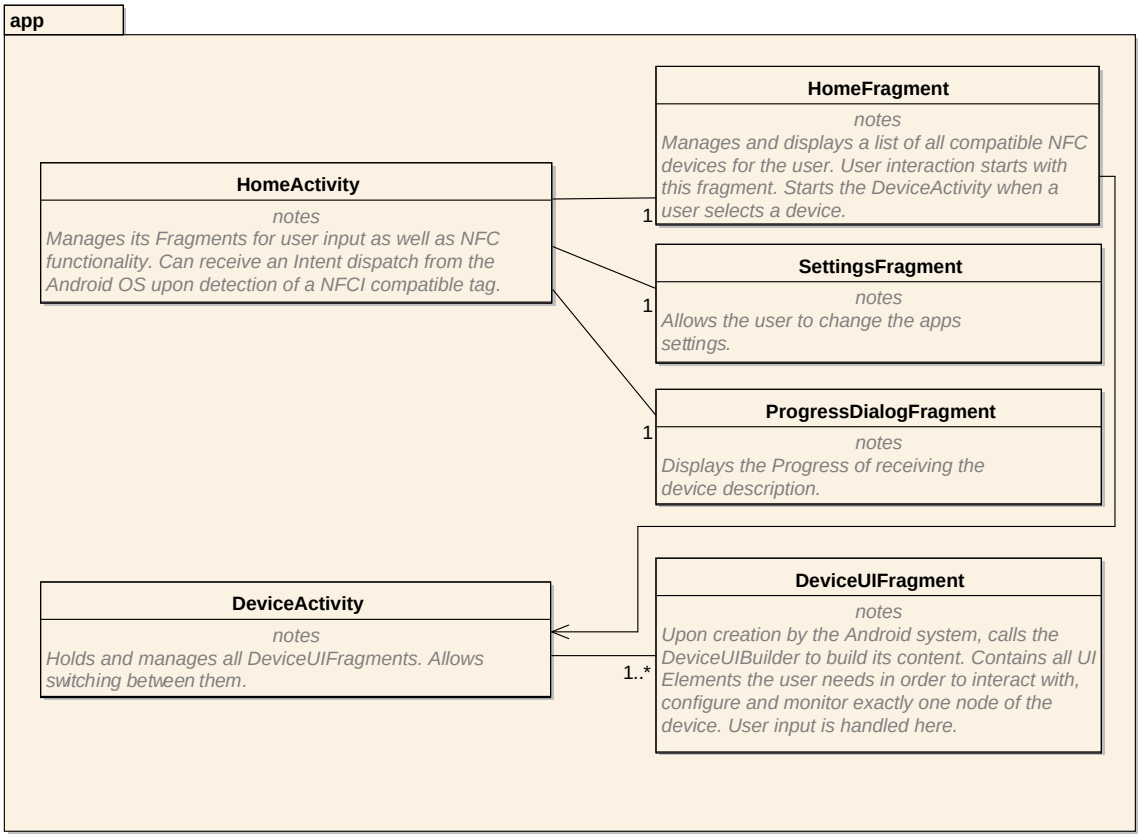

Figure 3.10: NFC Interface app class diagram, app package

HomeActivity: The *HomeActivity* represents the entry point and main screen of the app. It handles the *HomeFragment* which displays devices that are available to be connected to, or have already been connected to in the past. The app preference screen is as well handled in this activity. Furthermore, this activity is responsible for any actions to be taken when the app is launched, e.g. on the detection of new NFC tags being dispatched by the Android system. The NFCI features a unique type name which is being sent with any NDEF message allowing tags to be identified as belonging to the app and, therefore, be dispatched to it automatically.

HomeFragment: The *HomeFragment* contains a list of all NFCI devices available to be connected to, or already were connected to in the past. See chapter 3.6.1 for details on how devices are identified. It also indicates whether a device is currently in range. Whenever the user touches a new NFCI device, it appears in this list. On choosing the device, the smartphone downloads the NDEF message containing the TDD, creates and

displays the GUI for the Target Device to the user. If the description is already available on the smartphone, the download step will be bypassed.

DeviceUIActivity: The *DeviceUIActivity* is responsible for displaying all necessary *DeviceUIFragments*. This activity allows switching between the available *DeviceUIFragments* in an intuitive way, without any reload times.

DeviceUIFragment: As fragments are far more flexible than activities, it is possible to dynamically create multiple fragments for Target Devices with multiple Nodes. Each fragment will then be displayed as an independent *Fragment* within the *DeviceUIActivity* on the Android smartphone. As these fragments directly represent node functionality, the UI has to be built according to the TDD. The UI creation is handled by the *DeviceUIBuilder*.

SettingsFragment: The *SettingsFragment* will handle any kind of settings the user shall be able to define for the app globally. Such a setting might, for example, be the update interval of the NFCI status.

ProgressDialogFragment: This fragment displays the progress of receiving the TDD from the Target Device and creating the GUI. Depending on the TDD's size, this might take a few seconds (or longer) to retrieve, so this fragment ensures that the user gets some feedback at all times.

Communication Library and Tasks

The "communicationLibrary" package and its child package "tasks", both depicted in Figure 3.11, are responsible for any functionality related to communicating with the Target Device, as well as parsing the TDD.

TagCommunicator: The purpose of the *TagCommunicator* is to allow communication with the NFCI system. The functionality includes receiving NDEF messages containing the Device Identification and the TDD as well as enabling communication to the nodes.

CommunicationSerializer: The *CommunicationSerializer* encapsulates the creation of all commands to be sent to the Target Device. It is the central instance responsible for creating C-APDUs and ensures that changes to the communication protocol can be handled here.

XMLParserTask: The *XMLParserTask* parses the given XML-based TDD and creates the data structures representing the device. The XML Parser is not responsible for detecting errors within the TDD. In many cases, the parser may fail upon an error in the input, however this is not guaranteed.

NDEFReceiveTask: The *NDEFReceiveTask* allows receiving an NDEF message from the Target Device, for example, the NDEF message containing the Target Device Description.

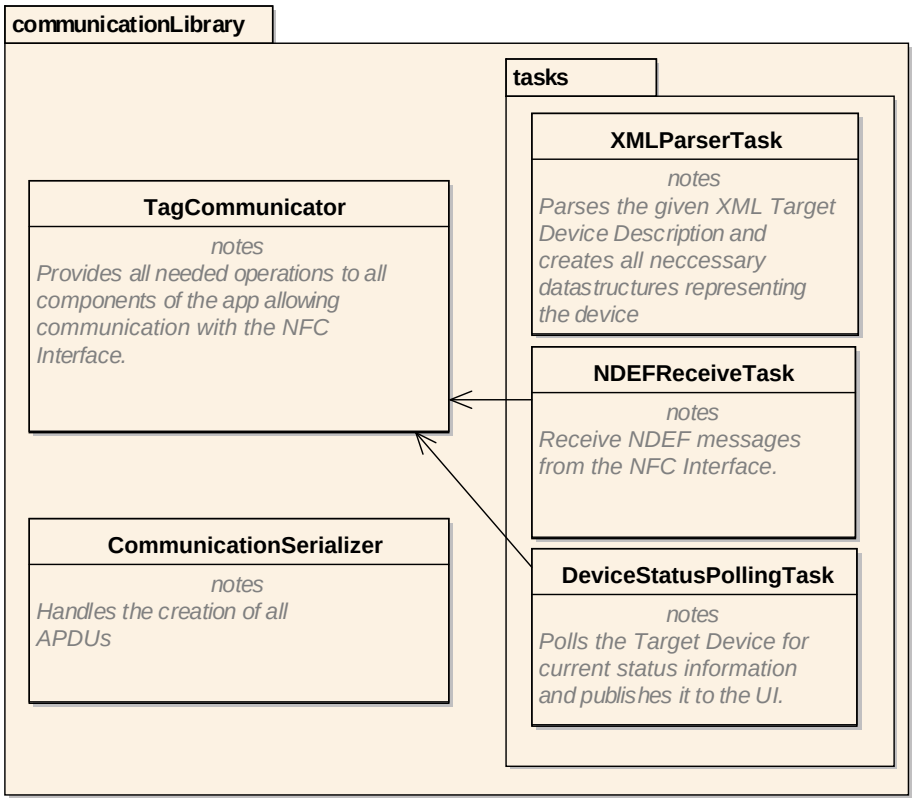

Figure 3.11: NFC Interface app class diagram, communicationLibrary package

DeviceStatusPollingTask: As NFC communication is synchronous, every C-APDU has exactly one R-APDU. The tag can only respond to commands from the reader, so the only possibility to get the current status from the Target Device is by polling, which is done by this class. The only alternative would be to use buttons for the user to choose when to retrieve the Target Device status. However, this hardly seems like a good design choice. Users expect such behaviour to be working automatically, without having to push a button.

Device Representation

The Target Device functionality is represented by the device data structure depicted in Figure 3.12. It consists of an instance of the class *Device* on top, which contains one or more instances of the *Node* class that, in turn, contain one or more *Element* class instances. Every *Element* can finally contain an arbitrary number of *Parameter* class instances. The exact functionality of the datastructure is explained in the following subsections. The information provided here is from a UI viewpoint, for a Parser/XML viewpoint, be sure to check chapter 3.7. Note that the following data structures also contain a varying amount of textual information to be presented to the user.

DeviceListItem: The class *DeviceListItem* class contains the most general information about the device, especially the parameters explained in chapter 3.6.1 in order to allow

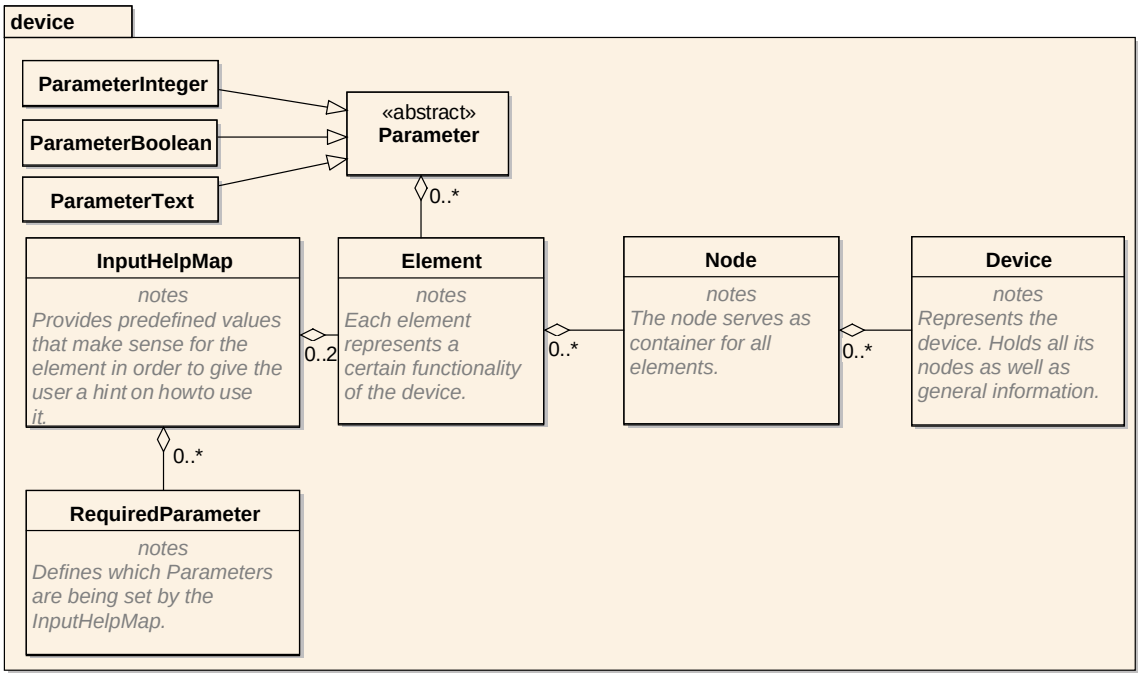

Figure 3.12: NFC Interface app class diagram, device package

unique identification of every device without the need to actually parse the TDD. This class is used to display very general device information within the *HomeFragment* and must be created upon receiving a new NFCI-compatible NFC tag via an intent from the Android system.

Device: The class *Device* serves as the link between the device data structure and the app functionality. It contains a reference to the *DeviceListItem* which is important in order to check whether a discovered device is equal to the one described. Furthermore, it holds a *Node* for every node description the TDD contains, which have to be provided upon creation of the *DeviceUIFragments*. The device, as well as the following data structures, is created upon parsing the TDD.

Node: A *Device* may contain one or more *Nodes*. A *Node* by itself does not describe any functionality, but serves as an encapsulation of functionality and provides an id which uniquely identifies the node on the Target Device.

Element: Every *Node* may contain one or more *Elements*. An *Element* represents some kind of functionality of the given node. It also provides an id uniquely identifying the element within the node.

Parameter: The *Parameter* class allows elements to be more configurable and useful. They serve the same purpose as parameters to a function. Every *Element* may contain an arbitrary number of parameters. Multiple parameter implementations will be created as necessary in order to represent String, Integer, Boolean and other types of parameters.

InputHelpMap: The *InputHelpMap* gives the user hints as on how to use a specific element. It can be seen as a "Suggestion List" and, furthermore, can automatically set all parameters available for an element according to the suggestion. Two types of *InputHelpMaps* are available, allowing users to either choose from a textual list or click on a grid of pictures. *InputHelpMaps* should only be supported within control and configuration elements.

RequiredParameter: The *RequiredParameter* class is linked closely to the *InputHelpMap* and contain a specific parameter value as well as its name. They allow the *InputHelpMap* to set a *Parameter* to defined values upon a user interacting with it.

UIBuilder

The "uiBuilder" package, depicted in Figure 3.13, is responsible for creating the GUI according to the device data structure whenever one is required.

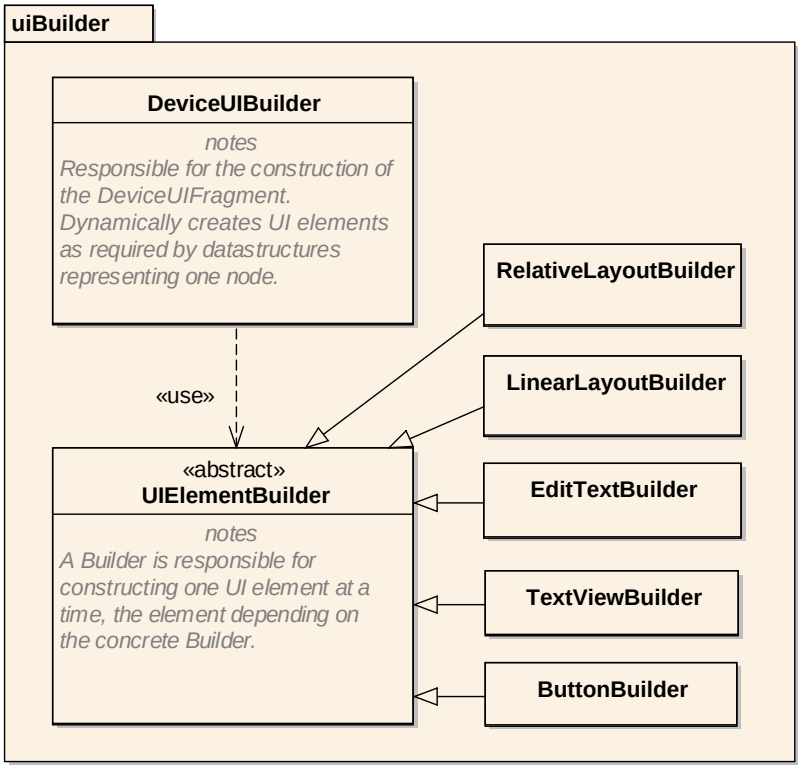

Figure 3.13: NFC Interface app class diagram, uiBuilder package

DeviceUIBuilder: The *DeviceUIBuilder* is the most important class of this package, coordinating the creation of the GUI for the connected Target Device according to the given data structure created by the *XMLParserTask*. The elements required for the user to interact with, control and monitor the Target Device are described in the TDD file. The *DeviceUIBuilder* builds view groups consisting of widgets like *TextView*, *EditText*, *Button*

or any other kind of UI widget required to fulfil the requirements. The widget‘s layout is achieved by using the *RelativeLayout* and *LinearLayout* classes for the basic alignment of widgets into view groups and layout parameters to align them. One view group displayed on the screen represents one Element specified by the TDD (see Figure 3.8). Every node is composed of multiple elements which represent specific functionality of the Target Device (e.g. cooking with the microwave). The *DeviceUIBuilder* creates the UI using the multiple available *UIElementBuilder* classes while traversing the data structure created by the *XMLParserTask*.

UIElementBuilder: The *UIElementBuilder* is the abstract base class for its subclasses. The implementations of the element builder are responsible for constructing one UI element at a time, depending on the concrete builder (e.g. for a *Button*, a *TextView*, an *EditText*, etc.).

3.6 Design – NFC Interface

3.6.1 Device Identification

Every device is uniquely identifiable via a UID. Different firmware versions for a device may require changes to the TDD. Also, the same TDD can be used by more than one device should they be identical. Therefore, using a unique identifier not only for the device itself, but also for the TDD is necessary. A simple hash of the TDD is utilized to achieve this. Using both identifiers ensures the TDD does not need to be sent via the (potentially slow) NFC channel if the very same file is already available.

3.6.2 Target Device Description

The TDD file has to be stored in the persistent memory of the NFCI while also allowing write access to the description so it can be changed after being deployed. For more information on the topic see chapter 3.7.

3.6.3 TagType4

TagType4 is a NFC tag communication specification by the NFC Forum. See the full specification [6] for details. The security controller firmware shall be extended to fully support TagType4 communication for selecting, reading and updating multiple NDEF messages. The following list gives an overview of the functionality required for TagType4 communication according to the standard:

- Selecting an NDEF tag application
- Selecting the capability container file
- Reading the capability container file
- Selecting the NDEF file
- NDEF read
- NDEF update

3.6.4 NDEF Messages

NDEF messages are the standard for exchanging information via NFC channels. Their headers include MIME types or other type definition formats identifying the type of the data contained within the NDEF message. See [For06a] for the exact specification of NDEF messages.

Every Target Device equipped with an NFCI must contain two NDEF messages. The first message contains the device's name, UID and a hash of the TDD. This message has to be the standard NDEF message for the TagType4 protocol, meaning it has to belong to the NDEF file identifier E104 in order to ensure it can automatically be detected by Android.

The second required NDEF message contains the TDD. It must be selectable via a TagType4 NDEF select command and must have a unique NDEF message ID, for example, E105. It can then be sent over the NFC channel, if necessary. In many cases, the TDD will already be stored on the device, so there is no need to exchange this data as this would only take up more time.

Further NDEF messages may be defined by the TDD in order to be read and written by the app. The Target Device supports reading and updating further NDEF messages according to the TagType4 standard.

3.7 Design – Target Device Description

In technical terms, the language used to describe the Target Device functionality is an XML-based, domain specific descriptive markup language. The term Target Device Description (TDD) shall emphasize the fact that a physical, electronic device's functionality is being described. In more detail the TDD has to be able to describe NFCI functionality as well as the nodes attached to it, including their respective control, monitor and configuration elements.

Monitor elements provide a way for the client to receive data from the device, control elements provide a way to send data to the device and configuration elements are capable of doing both by sending data to the device and receiving a response to this very request.

The XML schema every TDD follows is defined as an XML Schema Document ("XSD"). This offers a simple and effective way to determine whether the actual TDD is valid.

3.7.1 XML Structure

XML has been established to be the best choice for the task at hand. The XML-based TDD has to be able to describe any functionality the Target Device has to offer. XML follows a basic tree structure which can be used perfectly to describe Target Devices.

Figure 3.14 shows details on what the XML-based description must be able to describe and depicts the basic structure required. Code 3.1 conceptually shows an XML code fragment following the structure of Figure 3.14.

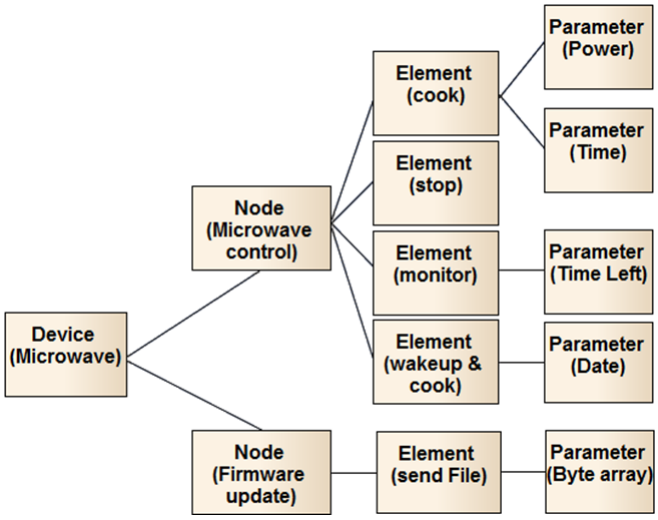

Figure 3.14: Basic XML description structure overview

Code 3.1 Example for the required XML structure

```
<device name="Microwave">
    <node name="MicrowaveControl">
        <element name="cook">
            <parameter name="Power">
    [...]
            </parameter>
       </element>
    [...]
    </node>
</device>
```

3.8 Design – Communication protocol

It would seem obvious to use XML not only for the specification, but also for the communication between the Target Device and the reader. However, this would introduce the need for a parser on the Target Device. Clearly, the advantage of not having to use a parser clearly is to reduce the risk of resource limitation on the Target Device (e.g. storage, time, computational power) as well as minimizing communication overhead.

The need for an additional parser can be avoided by using a sophisticated id system for all Target Device components. This means:

1. Nodes are grouped to a device and can be identified via a unique id (0-255).
2. Elements are grouped to a node and can in term be identified via another unique id (0-255).
3. Elements may contain an arbitrary number of different parameters as they will know their own parameters in advance.

Every Target Device element can now be addressed directly via these ids.

Commands are sent by the NFCI app to the Target Device in order to control it. The commands are sent in the form of a CAPDU (Command Application Protocol Data Unit), which is a byte array of a certain structure. Such a command might contain the information required to turn on a microwave. Sticking with the microwave example, it seems logical that the command mentioned above requires parameters, e.g., how much power to use, how long to stay active, whether to inform the user via a tone upon finish, or somewhat more exotic by sending a custom text message to the user.

To achieve the above, the system requires Boolean, Integer and String parameters at the least. The NFC enabled smartphone knows about these parameters as they are described in the TDD. The same goes for the Target Device since the description was created specifically for it.

As the TDD holds the ids for all nodes and elements, it is possible to identify any of these components uniquely. When a command is being sent to the Target Device, the id of the node as well as the one of the element has to be in the CAPDU payload. This allows the Target Device to directly relay any commands to the correct node without requiring any parsing. The parameters belonging to the element including their data types are defined within the TDD. String parameters need an additional byte indicating their length (max. 252 bytes long) while Integers are always four bytes long and Booleans are one byte long. The maximum length of the payload must not exceed 255 bytes.

Chapter 4

Implementation

This chapter describes the exact implementation of all NFC Interface (NFCI) system components according to the design described in chapter 3.

In detail, this chapter is divided into six sections. The first section presents the development environments. The second section deals with the implementation of the Android app. The following sections cover the specification of the Target Device Description (TDD), the implementation of the NFC Interface firmware and the Target Device applications. The final section deals with the testing methods used.

4.1 Development Environment

The implementation of the Android app was conducted using the Eclipse IDE[1] , Java SDK 7 and the Android SDK[2]. The used API version for the Android app is 15, corresponding to the Android operating system version 4.0.3.

The NFCI firmware was developed using the KEIL μVision IDE[3]. Deployment of the firmware via NFC was done with the Duali DE-620 contactless reader and the SmartCardManager software tool.

The Target Device PC applications were implemented using the most recent version of Microsoft Visual Studio 2012[4] and the Microsoft C# programming language. Furthermore, the XML Schema document for the TDDs was developed with Visual Studio.

4.2 Android App Functionality

4.2.1 Communication and Interaction

The *TagCommunicator* is the one instance handling any near field communication for the Android app. It has been extended in order to fully support TagType4 based communication with the Target Device. Native communication is still required to access node

[1] http://www.eclipse.org
[2] http://developer.android.com/sdk/index.html
[3] http://www.keil.com/uvision/
[4] http://www.microsoft.com/visualstudio

functionality on the Target Device. The *TagCommunicator* is a singleton class statically available to the whole app. It holds a reference to the actual tag object if one has been dispatched to the app by the Android system, and knows whether communication with the tag should currently be possible (tag is in range). The *TagCommunicator* supports two ways of communication with the NFCI:

TagType4 based communication: This part of the communication with the NFCI closely follows the standard defined by the NFC Forum in [For11]. Therefore, only an overview of the app's communication functionality is now given, without going into specific implementation details:

- Read the Device Information from the tag (provided as the standard NDEF message by Android corresponding to id E104 (see chapter 3.6.4 for details)).
- Read the TDD from the tag (corresponding to NDEF message id E105) via sending a set of APDUs representing the TagType4 communication protocol. Note this had to be done for the reason that Android does not support reading NDEF messages other than the one with id E104 through the default NFC API.
- Reads and write a third NDEF message via a set of APDUs representing the TagType4 communication protocol.

Native communication:

- Necessary for any command being sent in order to execute node functionality.
- Activation of the Target Device. Note that any native communication command sent to the Target Device via the *TagCommunicator* has previously been created by the *CommunicationSerializer*. The *CommunicationSerializer* takes parts of the device data structure, or other parameters and transforms them into CAPDUs. The following Code examples 4.1, 4.2 and 4.3 show the creation of some native commands used to communicate with the Target Device.

Code 4.1 Snippet for the creation of a command to read values from the Target Device

```
1  byte[] capdu = new byte[7];
2  capdu[0] = (byte) 0xC0;
3  capdu[1] = (byte) 0xF2;
4  capdu[2] = Constants.COMMAND_READ;
5  capdu[3] = (byte) 0;
6  capdu[4] = (byte) 2;
7  capdu[5] = (byte) nodeId;
8  capdu[6] = (byte) elementId;
```

Code 4.2 Snippet for the creation of a command to write values to the Target Device

```
1  int requiredSize = [...]
2  byte[] capdu = new byte[(requiredSize) + 7];
3  capdu[0] = (byte) 0xC0;
```

```
4    capdu[1] = (byte) 0xF2;
5    capdu[2] = Constants.COMMAND_SEND;
6    capdu[3] = (byte) 0;
7    capdu[4] = (byte) ((requiredSize) + 2);
8    capdu[5] = (byte) nodeId;
9    capdu[6] = (byte) elementId;
10
11   int i = 7;
12
13   for(Parameter currentParam : parameterList){
14
15     if(!currentParam.isSendable()){
16       continue;
17     }
18
19     if(currentParam.getClass().equals(ParameterInteger.class)){
20       capdu[i] = (byte) (((ParameterInteger)currentParam).getValue() >> 24)
             ;
21       capdu[i + 1] = (byte) (((ParameterInteger)currentParam).getValue() >>
              16);
22       capdu[i + 2] = (byte) (((ParameterInteger)currentParam).getValue() >>
              8);
23       capdu[i + 3] = (byte) (((ParameterInteger)currentParam).getValue());
24       i += 4;
25     } else if(currentParam.getClass().equals(ParameterText.class)){
26       capdu[i] = (byte) ((ParameterText)currentParam).getValue().length();
27       i++;
28
29       for(int j = 0; j < ((ParameterText)currentParam).getValue().length();
              j++){
30         capdu[i] = (byte) ((ParameterText)currentParam).getValue().charAt(j
               );
31         i++;
32       }
33     } else if(currentParam.getClass().equals(ParameterBoolean.class)){
34       capdu[i] = (byte) (((ParameterBoolean)currentParam).getValue() ? 1 :
             0);
35       i++;
36     }
37
38   }
```

Code 4.3 Snippet for the creation of a authentication command used to unlock restricted functionality on the Target Device

```
1    byte[] capdu = new byte[7 + password.length()];
2    capdu[0] = (byte) 0xC0;
3    capdu[1] = (byte) 0xF2;
4    capdu[2] = Constants.COMMAND_AUTHENTICATE;
5    capdu[3] = (byte) 0;
6    capdu[4] = (byte) (password.length() + 2);
7    capdu[5] = (byte) nodeId;
8    capdu[6] = (byte) elementId;
9
10   byte[] passwordBytes = password.getBytes();
```

```
11    System.arraycopy(passwordBytes, 0, capdu, 7, password.length());
```

4.2.2 Communication Scenarios

The Android app communicates with the Target Device upon certain user input. The specific communication scenarios are mapped to their preceding user input by table 4.1.

User input	**Resulting system action**
Touch a compatible tag	Receive device information
Click on a device on the main screen	1. *DeviceDescriptionReceiveTask* (if neccesary)
	2. Activate Target Device
	3. Start *DeviceStatusPollingTask*
Click on the activation symbol	1. Activate Target Device
	2. Start*DeviceStatusPollingTask*
Click on an NDEF read/write button in the Device GUI	1. NDEFMessage- Read/Write- Task
	2. Start *DeviceStatusPollingTask* (if neccesary)
Click on any other button in the Device GUI	1. Any native communication command
	2. Start *DeviceStatusPollingTask* (if neccesary)

Table 4.1: NFC Interface app communication scenarios

An important aspect of the app's communication protocol is the difference between the elements of a node. As described in chapter 3.7, three different element types exist: control elements, configuration elements and monitor elements. They differ in several ways starting with their functionality within the TDD, the way the UI is created for them and also in the structure of their respective APDUs. Figure 4.1 summarises the differences in the communication protocol for each element type.

4.2.3 Communication Tasks

Several functionality logically belonging to the *TagCommunicator* was split into a number of classes implementing *AsynchTasks*. These are mostly tasks that require more than one CAPDU to be sent which should be done outside of the main UI thread in order to ensure that the app stays responsive. Also, it allows tasks to be cancelled easily (e.g., if the user decides something is taking too long).

The **DeviceStatusPollingTask** polls the most recent values of all monitor elements for the currently displayed node from the tag. It is initialized upon the user opening a GUI for a device (except when he calls it via the "Show UI" context option). The task is running as long as the tag is in range. It can be reinitialized by pressing the Activate menu item in the Action Bar which activates the device if it is in range.

The **NDEFMessageReadTask/ NDEFMessageWriteTask** classes implement reading and writing an NDEF message with a certain id from or to a tag.

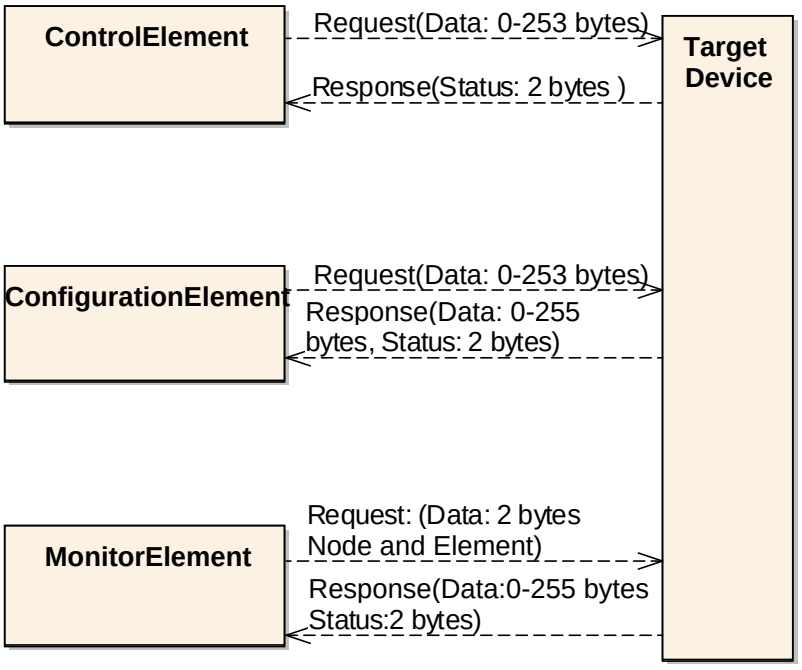

Figure 4.1: Requests from and responses to the different element types

The **DeviceDescriptionReceiveTask** class allows receiving the TDD from a tag. In differs from the *NDEFMessageReadTask* as it updates the *ProgressDialogFragment*, giving feedback to the user waiting for the download to be finished, as well as storing the TDD on the file system persistently.

4.2.4 Status of the Devices

On the app's main screen, there is a light indicating whether a device is in range or not. Once a device is close enough, it turns green and is red otherwise.

When opening the UI for a device, the Action Bar contains two menu items indicating the device's status. The first one is the "Activation" item. It displays an open lock symbol when the device is active and a closed lock symbol otherwise. The device being active means that communication to its nodes is possible. Inactive means only the NFCI can be communicated with, but the nodes are locked. The activation process can be done and undone by clicking on the "Activation" item.

The second item is called the "Online status" which shows whether the device is in range and can be communicated with. This is purely designed to keep the user informed. Figure 4.2 shows the Action Bar for a device that is both activated and online.

Figure 4.2: The NFCI app Action Bar indicating the devices status

4.2.5 XML Parsing

The XML parser is based on the *XMLPullParser*[5] available through the Android API. The *XMLPullParser* allows the code written for parsing the TDD to closely follow the structure of the actual code in a recursive way. The most important methods and concepts of this parser are explained in the following paragraph.

Most importantly, the parser always has a certain position within the XML structure. This position may be modified using its *nextTag()* method in order to find the next start or end tags, or *next()* to find the next parse event in general. The parser's current position may be assured using the *require(int type, String namespace, String name)* method. The method *getAttributeValue(String namespace, String name)* allows to read attributes, the method *getText()* returns the text content of the current element.

In detail, the parser recursively iterates through the XML structure starting with the <device> tag and creates the corresponding device datastructure step by step. An exact overview of the parser's functionality and execution order is given here:

1. Detect <device> tag.
2. Read <version>, <name>, <description> tags.
3. For every node available, read its attributes and:
   (a) For every device element within the node:
      i. Decide its type (control, configuration, monitor)
      ii. Read its elements, attributes and for every parameter:
         A. Read its attributes and elements
   (b) Read nested control elements (only for control elements)
   (c) Read user input aid (only for control and configuration elements)

An example of how the parser works is given in Code 4.4 which parses a monitor element and creates the corresponding datastructure. The code shows how the attributes and elements are parsed if they are available (Lines 2-19), including how the parameters representing the next lower level of the XML structure are read in Lines 21-25. Finally, the datastructure is created (Lines 27-31).

Code 4.4 XML Parser reading an element and creating its datastructure

```
1  private void readMonitorElement(XmlPullParser parser, ArrayList<Element>
       monitorElements) throws IOException, XmlPullParserException {
2    String monitorElementName = parser.getAttributeValue(namespace, NAME_TAG)
       ;
3    String monitorElementType = parser.getAttributeValue(namespace, TYPE_TAG)
       ;
4    int monitorElementId = Integer.parseInt(parser.getAttributeValue(
       namespace, ID_TAG));
```

[5] http://developer.android.com/reference/org/xmlpull/v1/XmlPullParser.html

```
5    boolean combine = false;
6
7    parser.nextTag();
8    String securityLevel = checkSecurityLevel(parser);
9    String text = checkForTextTag(parser);
10
11   if(parser.getName().equals(COMBINE_ELEMENT_TAG)){
12     if(TRUE.equals(readText(parser))){
13       combine = true;
14     }
15     parser.nextTag();
16   }
17
18   String currentElementTag = parser.getName();
19   ArrayList<Parameter> monitorElementParameters = new ArrayList<Parameter
         >();
20
21   while(currentElementTag.equals(PARAMETER_TAG)){
22     Parameter current = readParameter(parser);
23     monitorElementParameters.add(current);
24     currentElementTag = parser.getName();
25   }
26
27   Element monitorElement = new Element(monitorElementName,
         monitorElementType, monitorElementParameters, monitorElementId, false)
         ;
28   monitorElement.setText(text);
29   monitorElement.setCombine(combine);
30   monitorElement.setSecurityLevel(securityLevel);
31   monitorElements.add(monitorElement);
32 }
```

Note that the parser does not currently guarantee the detection of errors in the TDD. For example, using the same id more than once within its scope will not cause an error, but may lead to undefined behavior of the UI or the Target Device.

4.2.6 Device Representation

As mentioned before, the device data structure is created by the parser while the TDD is being parsed. It is an exact representation of the TDD based on which the graphical user interface (GUI) is built. Furthermore, data for the user can be read and stored in it. As this is a straight forward implementation of mostly container classes, no details beyond the ones given in the design chapter 3.5.2 will be given here.

4.2.7 Creating the Graphical User Interface

The most important part of this practical work is to provide the user with a GUI to interact with the Target Device. The process of its creation will now be explained in more detail. After the device data structure has been created by the XML parser, the GUI has to be built upon the information it contains. Highly abstracted, the GUI creation process goes through the following steps:

1. The *FragmentPagerAdapter* owned by the *DeviceActivity* is informed of the number of nodes for the device, and creates a Section for each one.
2. Upon the creation of each fragment's View, the basic layout is created and the following is done:
   (a) For every control element, create the layout and do the following:
      i. If authentication is required, create the authentication view and hide the functionality view created afterwards.
      ii. Create the input aid if available.
      iii. Create input fields for all parameters.
      iv. Create the buttons used for communication.
   (b) For every configuration element, create the layout and do the following:
      i. If authentication is required, create the authentication view and hide the functionality view created afterwards.
      ii. Create the input aid if available.
      iii. Create input/output fields for all parameters.
      iv. Create the buttons used for communication.
   (c) For every monitor element, create the layout and do the following:
      i. If authentication is required, create the authentication view and hide the functionality view created afterwards.
      ii. Create output fields for all parameters.
3. The final *DeviceUIFragments* holding all necessary views are now ready to be displayed to the user.

The most important concern for the UI creation process is the layout. The UI creation process offers an unlimited amount of possible GUIs. Therefore, the process of how the objects on the screen are aligned is a very important issue. The Figures 4.3 and 4.4 give an overview of the different layouts used and how they align their contained objects. In these Figures, the following color code was applied:

- **Blue**: Shows the borders of each outer RelativeLayout (representing one *Element* each).
- **Red**: Shows the borders of RelativeLayouts used for authentication (replacing the functionality of the elements until the user is authenticated).
- **Orange**: Shows the borders of all LinearLayouts.
- **Green**: Shows the borders of each inner RelativeLayout.

The meaning of each layout is explained in more detail now. Each element has an outer RelativeLayout. Contained within this layout is a text explaining the elements functionality to the user (if available) and the outer LinearLayout. This layout aligns the contained elements horizontally and in turn contains two more LinearLayouts, the so called inner

Figure 4.3: Layouts in the microwave oven GUI

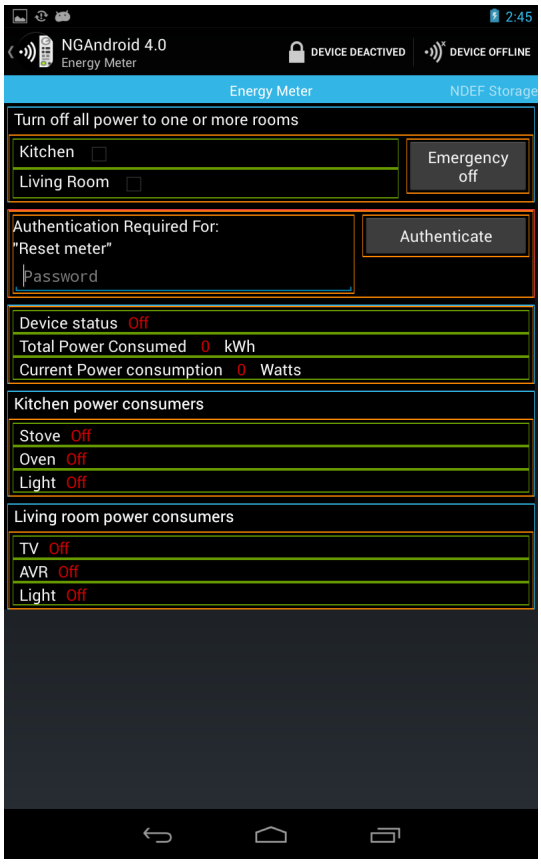

Figure 4.4: Layouts in the energy meter GUI

group layouts which align their content vertically. The right inner group layout is composed of all buttons while the left inner group layout contains a set of RelativeLayouts that in turn consist of most important UI components, such as input helps and parameters.

As an example, Code 4.5 shows the creation of an authentication group for the GUI. The creation process includes the construction of:

- The outer RelativeLayout (Line 3)
- The outer LinearLayout (Line 5)
- The inner group LinearLayout (Line 6)
- The widgets into the corresponding layouts
  - A TextView containing the string “Authentication Required For:” (Line 8)
  - A TextView containing the name of the element (Line 9)
  - A TextView containing an optional name for the security level if given (Lines 12, 13)

- An EditText representing the password input field (Line 16, 17)
- The Button allowing the user to start the authentication process (Line 18)

Code 4.5 Creation of an authentication group for the GUI including the creation of its widgets

```
1  private AnimatedRelativeLayout buildAuthenticationGroup(Element
       currentElement) {
2
3    currentLayout = AnimatedRelativeLayoutBuilder.buildMeInAnchorLayout(
         currentLayout, true, AnimatedRelativeLayout.FADE_RIGHT);
4
5    LinearLayout linearParent = LinearLayoutBuilder.buildMeInCurrentLayout(
         LinearLayout.HORIZONTAL, true, 0);
6    LinearLayoutBuilder.buildInnerGroupList(linearParent, 1.0f, 2.0f);
7
8    TextViewBuilder.buildMeInLeftInnerGroupLayout(getActivity().getResources
         ().getString(R.string.require_authentication));
9    TextViewBuilder.buildMeInLeftInnerGroupLayout("\"" + currentElement.
         getName() + "\"");
10
11   if(!currentElement.getSecurityLevel().equals("")){
12     TextViewBuilder.buildMeInRightInnerGroupLayout(getActivity().
           getResources().getString(R.string.authenticate_security_level));
13     TextViewBuilder.buildMeInRightInnerGroupLayout("\"" + currentElement.
           getSecurityLevel() + "\"");
14   }
15
16   EditText passwordField = (EditTextBuilder.buildMeInLeftInnerGroupLayout(
         getActivity().getResources().getString(R.string.password), true));
17   EditTextBuilder.setInputTypeToPassword(passwordField);
18   ButtonBuilder.buildMeForAuthenticationInCurrentLayout(currentElement, 0,
         passwordField, currentLayout);
19
20   getLeftSideDynamicViewsList().add(currentLayout);
21
22   return (AnimatedRelativeLayout) currentLayout;
23 }
```

Creation of Widgets

The creation of each widget, as shown in Code 4.5, is done via a one of the *UIElement-Builder* classes. The methods provided by these classes used to build the widget always follow the schema buildMe.* and, furthermore, contain information on which layout the widget should be inserted into. Note this was done instead of the desired layout being just a parameter to one buildMe(...) method since not every element can be built into every layout which could really only be explained by comments. Therefore, different method names seemed easier to maintain for the future.

The Code example 4.6 explains how the widget is aligned into a specified layout. The *DeviceUIBuilder* holds a number of members allowing it to identify the most recently created layout (The *currentLayout*, which corresponds to the inner RelativeLayouts depicted

in Figure 4.3) as well as lists of all views matching certain criteria.

Code 4.6 Creation of an EditText via the EditTextBuilder and placement into the current layout

```
1  public static EditText buildMeInCurrentLayout(String hint, int
       horizontalOffset, boolean layout, boolean restrictLength){
2    DeviceUIBuilder builder = DeviceUIBuilder.getInstance();
3
4    EditText editText = buildMeFree(hint, horizontalOffset, layout,
         restrictLength);
5
6    builder.addToLeftViewList(editText);
7    builder.addToCurrentElementLayout(editText);
8
9    return editText;
10 }
```

Code 4.6 does not show the actual process of creating the widget since this is done by the method *buildMeFree(...)* as shown in Code 4.7. The code first creates and formats the EditText itself (Lines 4-7), then restricts its length to the size of its hint if requested (Lines 11-16) and, finally, attaches the RelativeLayout LayoutParameters if applicable (Lines 19-21).

Code 4.7 Internal creation of the EditText in the EditTextBuilder without being attached to a layout

```
1  private static EditText buildMeFree(String hint, int horizontalOffset,
       boolean layout, boolean restrictLength){
2    DeviceUIBuilder builder = DeviceUIBuilder.getInstance();
3
4    final EditText editText = new EditText(builder.getActivity());
5    editText.setId(builder.getNextElementId());
6    editText.setHint(hint);
7    editText.setTextAppearance(builder.getActivity(), R.style.
         edit_text_appearance);
8
9    DisplayMetrics dm = builder.getDisplayMetrics();
10
11   if(restrictLength){ // calculate the width of the hint independant of
         screen sizes
12     Paint paint = new Paint();
13     paint.setTextSize(28 * dm.density);
14     final float size = hint.length() >= 6 ? paint.measureText(hint) : paint
           .measureText("WWWWWW"); //widest letter
15
16     editText.setMaxWidth((int)size);
17   }
18
19   if(layout){
20     RelativeLayout.LayoutParams layoutParams = RelativeLayoutBuilder.
           prepareLeftSideLayoutParameters(horizontalOffset);
21     editText.setLayoutParams(layoutParams);
```

```
22     }
23
24     return editText;
25   }
```

4.3 XML Specification

The content of the XML file aims to describe all Target Device functionality which should be accessible remotely.

In order to avoid confusion between the XML concept of elements and the data structure of control, configuration and monitor elements, these elements will be referred to "device-elements".

Note that the order of elements in the XML file is important to the parsing process unless explicitly specified otherwise. Random order will make the parser fail to process the TDD. However, the order of attributes is not important, they can be interchanged without causing any side effects, unless explicitly specified otherwise.

The following is an overview, a set of rules, must and must not's when using this XML-based descriptive markup language in order to describe Target Device functionality. There are many ways to express the same kind of functionality, but there are no checks towards the meaningfulness of a TDD or even redundancy. This is entirely up to its creator. Also, the parser is not responsible for finding or failing on possible mistakes in the TDD, that's what the XML schema document (XSD) is for. Using faulty TDDs with the app (for example, using the same id for multiple elements within a node) might cause undefined behavior within the parser, during UI creation, or at runtime! Faults concerning the general structure of the TDD will, however, almost certainly be detected while it is being parsed.

The general structure of the descriptions is now be explained in more detail:

- <version>: The version tag of the descriptive language used to create this TDD. This element has to be incremented according to the version this description was created with. Note it is not currently used and only available for future compatibility.
- <name>: the full device name shown to the user.
- <description>: a short textual description of the device.

The following element <node> defines a node available for interaction on the Target Device. The description may contain multiple nodes. Each node has to have two attributes:

- The *name* attribute which is displayed to the user.
- The *id* attribute which is identifying the node. Ids have to be used in an ascending order ($\forall id : id_{i+1} > id_i$) and they must be of value (0-255).

Each node may contain multiple device-element tags. A device-element tag and all of its content is displayed to the user as a coherent structure (see chapter 4.2.7 for details). Elements can be of the following types:

- <controlElement>: Defines functionality to be executed on the Target Device. These elements send data to the Target Device. They might depend on configuration or monitor elements (e.g. do not allow to open washing machine door while it's washing); control elements, however, do not contain data that is subject to change. The only response a control element gets from the Target Device is a two byte status from the NFCI. Any responses to a control element command belong into a monitor element.
- <configurationElement>: Defines functionality and values of the Target Device that are not subject to frequent change. Values within a configuration element may usually (but do not have to) be changed by the user. Furthermore, they can be changed by the Target Device as a response to the command being sent. That being said, responses are not limited to the two bytes as with control elements.
- <monitorElement>: Defines the current status of the Target Device and is subject to frequent change. These values are read-only; the reader cannot change them directly, but indirectly via control elements. Currently, they have to be polled by the reader to ensure frequent updates.

The device-elements contain the following required attributes:

- The *name* attribute, which is displayed to the user as a button in order to execute the functionality described.
- The *id* attribute offers a unique identification of the device-element within the current node. Again, ids have to be used in an ascending order ($\forall id : id_{i+1} > id_i$) and they must be of value (0-255).
- The *type* attribute which describes the way this element should behave, mainly in order for the app to decide whether it requires an input or just an output field. Possible types are:
  - Post: Sends data and includes all parameters belonging to the element. In a control element, the response only consists of two status bytes. In a configuration element, the response might contain additional data.
  - Monitor: Indicates that the user should have no possibility to modify the content shown by this element. The values will be updated through the DeviceStatusPollingTask.
  - Secure Post: Includes the same functionality as Post, however, prior to using the actual functionality, the user has to authenticate himself via a password.
  - Secure Monitor: Includes the same functionality as Monitor, however, prior to using the actual functionality, the user has to authenticate himself via a password.
  - ReadNDEF: Indicates this device-element is used to read an NDEF message from the NFCI. It requires the <nDEFMessage> element to be present, and have a parameter with an id attribute of value "0" and a unit attribute equal to "Text". As this functionality requires sending a command and receiving a response into the same device-element, it is only allowed within a configuration element.

  - WriteNDEF: Indicates this device-element is used to write an NDEF message to the NFCI. It has the same requirement as the ReadNDEF type above. This type is only allowed within a control element.

Every device-element may contain an optional <text> tag giving additional information about this element to the user. It is always displayed as a heading to the information contained within the device-element.

A device-element may also contain the <combineElements> tag. It allows information across multiple elements logically belonging together to be displayed as such by the UI. Note this can easily lead to confusion as to what a device-element does and does not do. Therefore, this tag should be used with care.

Special case: the <nDEFMessage> element specifies the ID of an NDEF Message as a hex string (e.g., "E104") which can be directly written and read. This tag is only allowed when the type of the element is ReadNDEF or WriteNDEF.

All elements can hold parameters allowing for more versatile functionality. The tag <parameter> permits definition of such parameters in order to present the user with required input possibilities as well as providing him information about the device's state. Every parameter requires additional information as follows:

- The *name* attribute, which identifies the parameter in a humanly understandable way, but does not have to be unique. The name is usually displayed to the user.
- The <id> element, which identifies the parameter and has to be unique for every parameter within the current element. Again, ids have to be used in an ascending order ($\forall id : id_{i+1} > id_i$) and they must be of value (0-255).
- The *type* attribute contained within the <id> element, defining the parameters type. The type can be "Immutable" for values that cannot be changed by the Target Device (usually values belonging to control elements), "Monitor" for values to be updated regularly or "Configuration" for configuration values. Note their functionality has been refactored away for now, therefore at this point the type attribute is not required and may be missing.
- The required <unit> element, which contains information on how the element is to be displayed to the user, this is usually something meaningful (Meters, Liters, Watts,...). The unit itself may be of three different types. Type "Boolean" means the attributes textTrue and textFalse can define the way this parameter is shown to the user in a monitor element. If the type is "Text" no further elements are necessary. Otherwise, the attributes minValue and maxValue can be defined and the internal representation of the parameter is an integer value.
- The optional element <visibility>, which can be set to "invisible" in order to hide the parameter from the UI. This may be used in combination with the userInputAid in order to send parameters invisible to the user to the Target Device. Note this element was chose not to be a boolean as it might offer additional functionality (e.g. highlighting) later on.

- Finally, the optional element <sendable>, which may be set to false in order to not send the parameter to the Target Device. This may be used in combination with the userInputAid to provide the user with information that is not needed by the Target Device.

The <nestedControlElement> tag offers the possibility to add more functionality to a control element as it logically belongs to its parent on the UI (e.g. turn on, pause, stop). They can be configured to use their parents' parameters, or no parameters at all, however, they may not contain additional parameter definitions. The <parameter> element required by a nested control element allows the values "None" in order not to use any parameters at all, or "Default" in order to actually use its parents' parameters.

The optional element <userInputAid> (found in control and configuration elements) can be used to display additional information to the user, for example, a list with possible input. The userInputAid can access all parameters defined within the current device-element tag and set them to a value. It relates to the InputHelpMap class. The following attributes are required for this element:

- The *name* attribute, which contains a name to be displayed to the user for this input aid (e.g. "Choose Mode").
- The *type* attribute, identifying the type of the userInputAid. Two types are available, "List", displaying a dropdown list of possible items to the user, and "Grid", displaying a grid of images to the user.

Note that depending on the userInputAid type, different elements are required.

- List: This type requires elements named <listItem> containing the name of the list item (e.g. "Boil Water" in the microwave example). Every <listItem> can contain a number of attributes with their names corresponding to the parameter they describe. Its value will be assigned to the parameter upon a user clicking in the list.
- Grid: This type requires elements named <icon> consisting of the same attributes as the <listItem> defined above. However, the <icon> element does not contain items' names. Instead, it includes an additional element called <name>. This element in turn has to contain an attribute named imgsrc, defining the location of the image to be displayed relative to the /NGAndroid folder within the Android external storage directory. The image must already be stored on the Android device and can currently not be transferred from the NFCI device.

Example TDD files for two prototypes, a microwave oven and an energy meter can be found in the Appendix, Section A.1 and A.2). An XML schema document (XSD) was created allowing a quick check of newly created TDDs. The schema does not prove the description to be absolutely correct. For example, the values of id attributes are not checked to be without faults (unique within one element and in ascending order). However, using the schema to check the basic structure of a new TDD is highly recommended.

4.4 NFC Interface

The NFCI firmware was modified in order to fully support TagType4 based communication. The communication channels in general (native communication to the nodes) and the hardware setup of the NFCI remain unchanged. A detailed description of the NFCI implementation can be found in [Bas13].

4.4.1 Device Identification

Every device contains a default NDEF message identifying it as an NFCI device. The NDEF message header contains a definition of "NFC Forum External Type", allowing the message to be forwarded directly to the NFCI device. The name of this type is "at.infineon:NFCI". The NDEF message includes data specific to the Target Device, a name, a UID and a hash of the TDD file. This ensures that every device can be identified uniquely and independent of its TDD. Different devices can use the same TDD.

This allows the NFCI smartphone app to decide whether a download of the current TDD is necessary or an identical, already stored, description is available. The following example shows the first NDEF message of an Energy Meter: "NAME:Energy Meter UID:190091 HASH:4cf9b8"

4.4.2 Device Description

The TDD is stored in the NFCI's non volatile memory and can be downloaded as an NDEF message with the id E105 via the TagType4 protocol at any time. The NDEF message header is the same as for the DeviceIdentification described above.

4.4.3 TagType4 Communication

As indicated in chapter 3.6.3, the TagType4 protocol has been implemented. The currently supported operations include:

- Selecting an NDEF tag application (default application D2760000850101, as defined in [For11])
- Selecting the capability container file (default container with id E103, as defined in [For11])
- Reading the capability container file
- Selecting the NDEF file (default message with id E104, as defined in [For11]: Device Identification; E105: Target Device Description; E106: configurable custom NDEF message)
- NDEF read (all NDEF messages)
- NDEF update (NDEF message with id E106)

4.5 Target Device Applications

The Target Device PC applications are connected to the NFCI acting as nodes. They are based on the work done by [Bas13]. Their basic design and structure stay the same. They were slightly modified in order for them to support interaction with the new Android app, mostly because of the new communication commands which have been described in chapter 4.2.1. The communication path between the nodes and the NFCI was not changed.

4.6 Testing the Android App

Unit tests were implemented using the JUnit based Android testing framework[6]. While it is not ensured at this point that any functionality actually has one or more test cases, the important parts do. The tests were implemented using Android Unit Tests, Java Reflection[7] and special technologies for two important issues, ensuring any functionality can actually be tested without requiring any NFC communication and a Target Device.

4.6.1 User Interface Testing

The Robotium[8] UI testing framework allows runtime testing of Android user interfaces. It can be used to simulate clicks, check the existence or interact in other ways with any UI component. This is very important as some functionality can only be executed depending on input from the UI and, therefore, can only be tested with a UI in place. The following Code 4.8 shows how Robotium is used to ensure the right values are displayed on the UI after the user clicks on a spinner item (in the microwave oven GUI). Robotium clicks on the spinner (Line 7) in order to display the spinners dropdown list. Then it clicks on an item displayed in the list (Line 8) and, finally, ensures the correct values have been set on the UI (Lines 10, 11). The same is then done another time with different spinner items.

Code 4.8 Robotium being used to click a spinner and check its output on the UI

```
1  public void testSpinner() {
2    startDeviceUIFragment();
3
4    EditText duration = [...];
5    EditText power = [...];
6
7    solo.clickOnText("Boil");
8    solo.clickOnText("Heat");
9    solo.sleep(100);
10   assertTrue(duration.getText().toString().equals("240"));
11   assertTrue(power.getText().toString().equals("400"));
12
13   solo.clickOnText("Heat");
14   solo.clickOnText("Defrost");
15   solo.sleep(100);
16   assertTrue(duration.getText().toString().equals("480"));
```

[6] http://developer.android.com/tools/testing/testing_android.html
[7] http://docs.oracle.com/javase/tutorial/reflect/
[8] http://code.google.com/p/robotium/

```
17    assertTrue(power.getText().toString().equals("80"));
18
19    [...]
20  }
```

4.6.2 Mockup Testing

The mock framework Mockito[9] is used for mockup tests. In short, it allows to replace objects by mocks, which are basically fake objects and do not contain any functionality. However, these fakes know when their methods are being called, can record parameters and return values defined in the test case. In the NFCI Android app it is mainly used to test functionality dependant on NFC and make parts of the app act as if a NFC device is actually connected.

The example Code 4.9 creates a mock of the *TagCommunicator* (Line 2), sets it as it's singleton instance (Line 4) via a helper method using Java Reflection, captures the arguments and sets the return values of some important TagCommunicator methods (Lines 8, 9).

Code 4.9 Mockito mocking the TagCommunicator

```
1  private void createTagCommunicatorMock(){
2    mockCommunicator = mock(TagCommunicator.class);
3
4    ReflectionUtils.setPrivateField(TagCommunicator.getInstance(), "instance
         ", mockCommunicator);
5
6    authenticationArgument = ArgumentCaptor.forClass(byte[].class);
7
8    when(mockCommunicator.sendCapduForByteResponse(authenticationArgument.
         capture())).thenReturn(Constants.ISOSW_SUCCESS);
9    when(mockCommunicator.checkSuccessfulResponse(authenticationArgument.
         capture())).thenReturn(true);
10 }
```

Upon a UI test now clicking on a button, the success status bytes will be returned by the mocked TagCommunicator, while no actual NFC communication is invoked.

This very functionality can be now be used to test any communication scenario, for example, unlocking an element via its authentication view, which is shown in Code 4.10. In detail, first, references to the elements about to be unlocked are being retrieved (Lines 8, 9) and it is being ensured that only the authentication views are visible (Lines 11 - 14). Note that both these elements belong to security level red, so unlocking either one unlocks the other as well. Afterwards, Robotium is used to simulate a click on the Authenticate button (Line 16). At this point, the mocked *TagCommunicator* has returned a success response allowing the app to unlock the functionality belonging to the element's security level. The next step in the test case is to check that both elements have been unlocked

[9] http://code.google.com/p/mockito/

(Lines 19-22). Finally, the argument passed to the mocked *TagCommunicator* is checked to correspond to the defined communication protocol (see Chapter 4.2.1 for details) (Lines 24 - 26).

Note that even though the element can be unlocked this way, its functionality could not be executed by the Target Device as it did not actually approve this authentication.

Code 4.10 Mockito and Robotium being used to test a communication scenario without invoking NFC functionality

```
1  public void testUnlockAuthentication(){
2    createTagCommunicatorMock();
3    startDeviceUIFragment();
4
5    DeviceActivity deviceActivity = (DeviceActivity) solo.getCurrentActivity
         ();
6    Device device = (Device) ReflectionUtils.getPrivateField(deviceActivity,
         "device");
7
8    Element red1 = device.getNode(0).getElement(Node.ElementType.
         controlElement, 1);
9    Element red2 = device.getNode(0).getElement(Node.ElementType.
         monitorElement, 4);
10
11   assertTrue(red1.getAuthenticationGroup().getVisibility() == View.VISIBLE)
         ;
12   assertTrue(red1.getFunctionalityGroup().getVisibility() == View.GONE);
13   assertTrue(red2.getAuthenticationGroup().getVisibility() == View.VISIBLE)
         ;
14   assertTrue(red2.getFunctionalityGroup().getVisibility() == View.GONE);
15
16   solo.clickOnText(solo.getCurrentActivity().getResources().getString(at.
         infineon.ngandroid4.app.R.string.authenticate)); //unlock red
17   solo.sleep(500);
18
19   assertTrue(red1.getAuthenticationGroup().getVisibility() == View.GONE);
20   assertTrue(red1.getFunctionalityGroup().getVisibility() == View.VISIBLE);
21   assertTrue(red2.getAuthenticationGroup().getVisibility() == View.GONE);
22   assertTrue(red2.getFunctionalityGroup().getVisibility() == View.VISIBLE);
23
24   List<byte[]> authParameters = authenticationArgument.getAllValues();
25   String authRequest = Utils.byteArrayToHexString(authParameters.get(0));
26   assertEquals("Sent command for Authentication not the one expected!", "
         C0F22000020101", authRequest);
27
28 }
```

Chapter 5

Results

This chapter gives an overview of the results achieved by this thesis' practical implementation. The main goal was to provide a graphical user interface (GUI) on Android smartphones for embedded devices through the usage of an XML-based descriptive markup language. In order to demonstrate all of the results, two use cases will illustrate the functionality provided by the NFC Interface (NFCI) system from two viewpoints: the NFCI app and the NFCI nodes (PC applications). These use cases are:

- Microwave oven use case
- Energy meter use case

5.1 Setup

The complete system setup includes all components described in the previous chapters. The hardware used to interface the Target Device, as mentioned before, was already developed prior to this thesis and, therefore, will not be explained in detail. Note that the NFCI app will be referred to as "NGAndroid 4.0" on all screenshots. This corresponds to the current internal versioning. Two Android devices are used as reference devices for the NFCI app: The Samsung Galaxy S smartphone, running a stock version Android 4.2.1, and the Asus Nexus 7 tablet, running the default operating system, Android 4.2.2.

Further testing has been conducted on the following smartphones, using their respective default operating system at its most recent version: LG Nexus 7, Samsung Galaxy S3 Mini, Samsung Galaxy S3, Samsung Galaxy S4, HTC Desire HD (GUI only, as no NFC module is available).

Figure 5.1 shows the exact setup used for the presentation of the results. It is divided into three parts: (upper left) The Galaxy S smartphone with the NFCI app displaying its home screen, (upper right) the NFCI hardware configured to contain the energy meter description and (lower right) the Target Device PC application. The arrows between the images show the communication path and their respective technology.

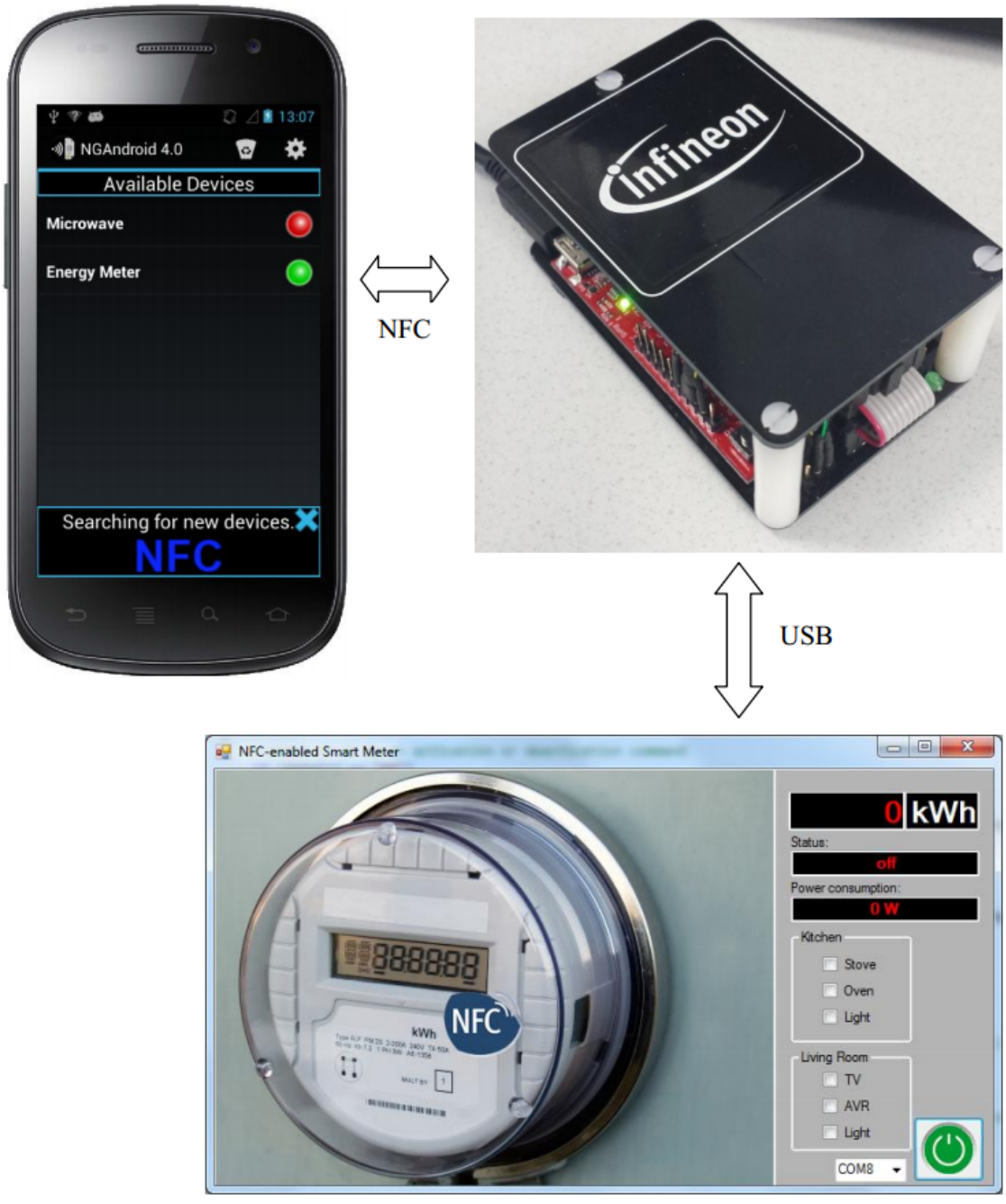

Figure 5.1: Overview of all parts of the prototype system: The Android app, the NFCI hardware and the Target Device software (adapted from [Bas13]) including their communication paths

5.2 Target Device Management in the NFC Interface App

In order to communicate with the energy meter or any other device, the user has to keep his smartphone close the NFCI antenna. The green light shown in Figure 5.1 indicates

the respective device is in range. In this case, the Target Device Description (TDD) is already available, so clicking on the device immediately parses the TDD and creates the corresponding GUI.

Should the device not have been connected to before (and its TDD is not already available through another device using the same one) it is necessary to download the TDD from the Target Device. In order to demonstrate the download of the TDD, the device has to be deleted. Figure 5.2 shows how the UI can be used to delete the device. Touching the tag again afterwards leads the UI to show that a new device has been detected as depicted in Figure 5.3. Upon clicking on the newly discovered device, the system needs to download the TDD. The Progress Dialog pops up at this point, indicating the status of the download of the NDEF message containing the TDD and continuously providing updates to the user.

Figure 5.2: UI for deleting one or more devices

Figure 5.3: UI showing a new device has been found

5.3 Target Device Description Download

The following values depicted in Tables 5.1 and 5.2 indicate the time required to download the TDD NDEF message from the NFCI. They include a small overhead as the received data continuously gets written into a byte buffer as well as the overhead caused by the Target Device firmware as it reads the data from its persistent memory. It does not include TagType4 specific overhead (e.g. select NDEF message). Each description was downloaded five times with the NFC antenna being in a slightly different position every time, as it would also happen in a real environment.

The average data transfer rate is 4,33 kB/s for the Device I and 3,2 kB/s for the Device II. The difference can be explained as a result of the relatively smaller overhead for the larger file caused by the NDEF message header.

Name of Target Device	**Description size in kB**	**Time in seconds**
Device I	5.07	1,141
		1,257
		1,110
		1,196
		1,148
Average time in seconds		1,170

Table 5.1: NFC-based Target Device Description download time for Device I

Name of Target Device	**Description size in kB**	**Time in seconds**
Device II	1.73	0,535
		0,570
		0,524
		0,531
		0,523
Average time in seconds		0,536

Table 5.2: NFC-based Target Device Description download time for Device II

A transfer time of potentially more than one second is definitely noticeable to a user. Therefore, the app contains a dialog displaying the progress of the TDD download.

5.4 Energy Meter Use Case

The energy meter is designed to keep track of a user's power consumption. In order to do this, energy consuming devices are a needed. In the implemented use case, the Target Device (PC application) is keeping track of a number of energy consuming devices distributed in two rooms: a kitchen and a living room. The energy consuming devices can either be active or inactive. When active they consume power, otherwise they do not as standby energy consumption is not implemented in the example.

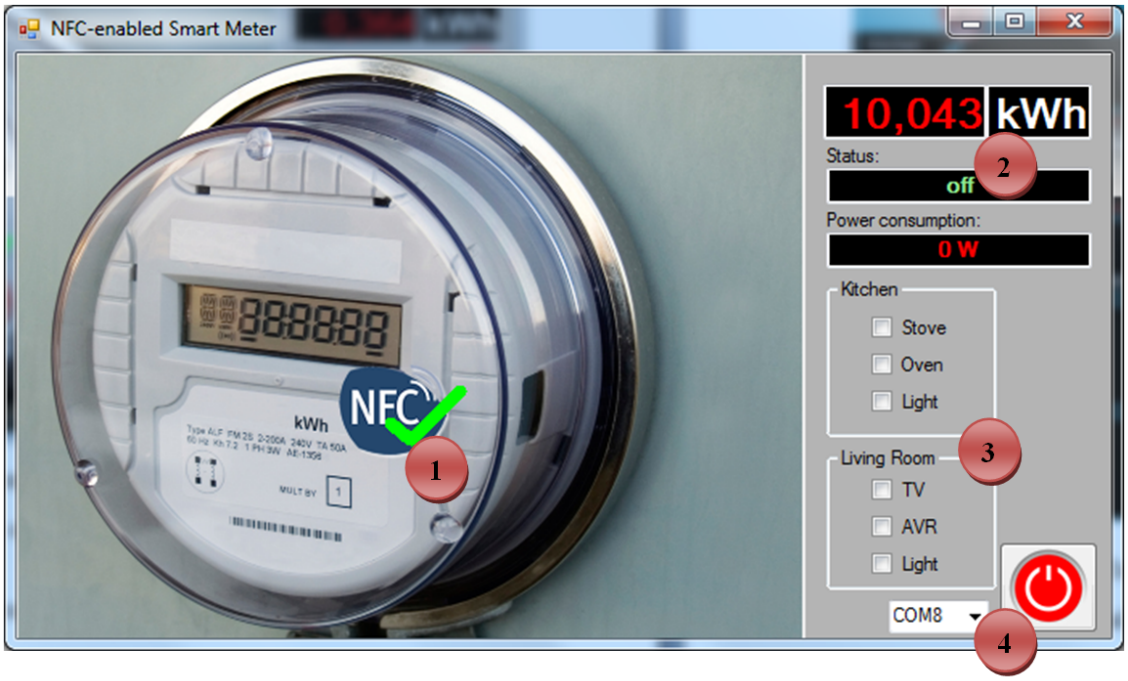

Figure 5.4: Energy meter PC application posing as a node attached to the NFC Interface (adapted from [Bas13])

5.4.1 Target Device PC Application

For demonstration purposes, the energy consuming devices can be switched on and off via a simple click on the PC application GUI. Note this is just a freely invented use case aimed at demonstrating the systems features and does not reflect the functionality of a real-world energy meter!

Figure 5.4 shows all functionality of the energy meter node. Its GUI components are now explained in more detail:

1. The NFC indicator, indicating a compatible smartphone has activated the Target Device.
2. Values indicating the status as well as the total and current power consumption monitored by the energy meter.
3. A list of power consuming devices and their status (on/off).
4. The power button and the serial port number used to connect the PC application to the NFCI.

5.4.2 Android App

The Android app GUI pictured in Figure 5.5 is built by the app based on the TDD. The full TDD can be found in Appendix A.1.

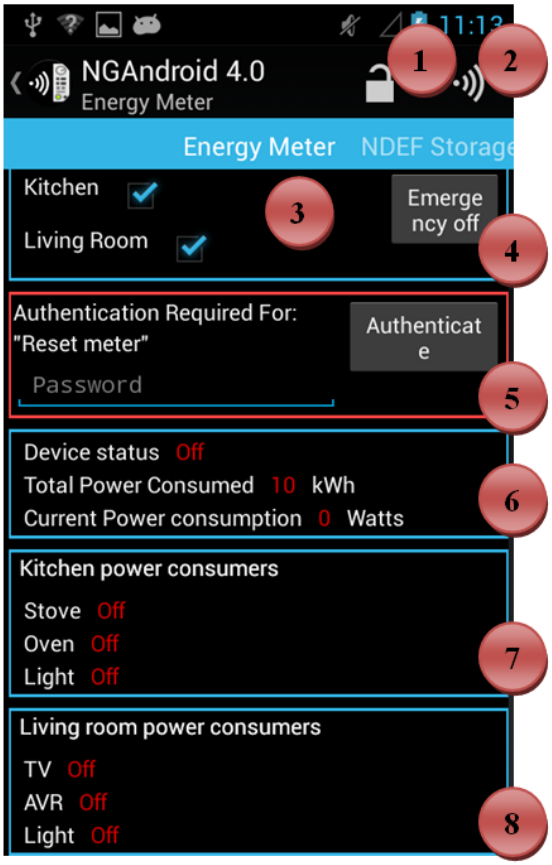

Figure 5.5: Android GUI for the energy meter

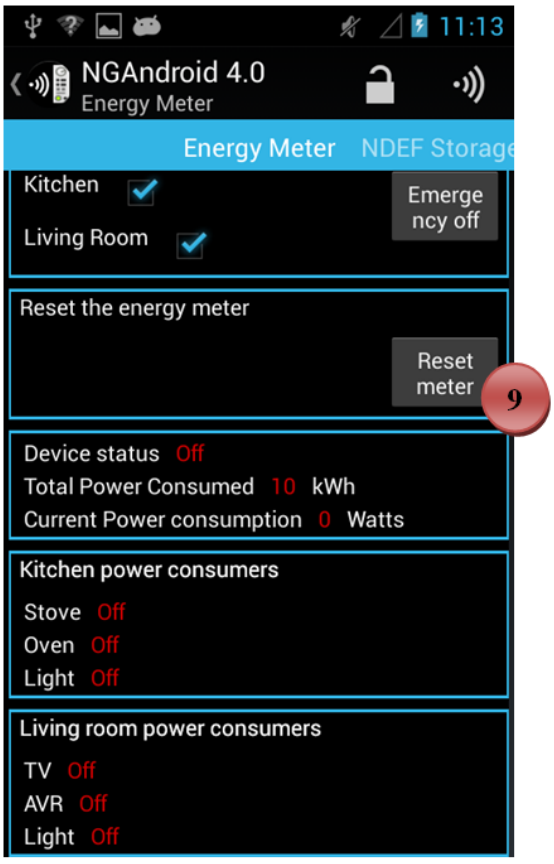

Figure 5.6: Android GUI for the energy meter after successful authentication

The functionality of the GUI shown and enumerated in Figures 5.5 and 5.6 will now be explained in more detail:

1. Activation status: Indicates whether the Target Device is activated. An activated Target Device allows communication to its nodes. Clicking on this item will activate/deactivate the Target Device.

2. Online status: Indicates whether the device is in range and can be communicated with.

3. The name of the current node, also indicating that other nodes are available.

4. Emergency off element: Allows shutting down power in the checked areas.

5. Authentication element: Allows the user to authenticate with a password in order to obtain access to the element's functionality.

6. Monitor the current energy consumption and total energy consumed.

7. Monitor active devices in the kitchen.

8. Monitor active devices in the living room; note that, since this element does not fit on the screen entirely, the page becomes scrollable.

9. After entering the correct password, the PC application has authenticated the user and the actual functionality has been unlocked. The user may now reset the energy meter.

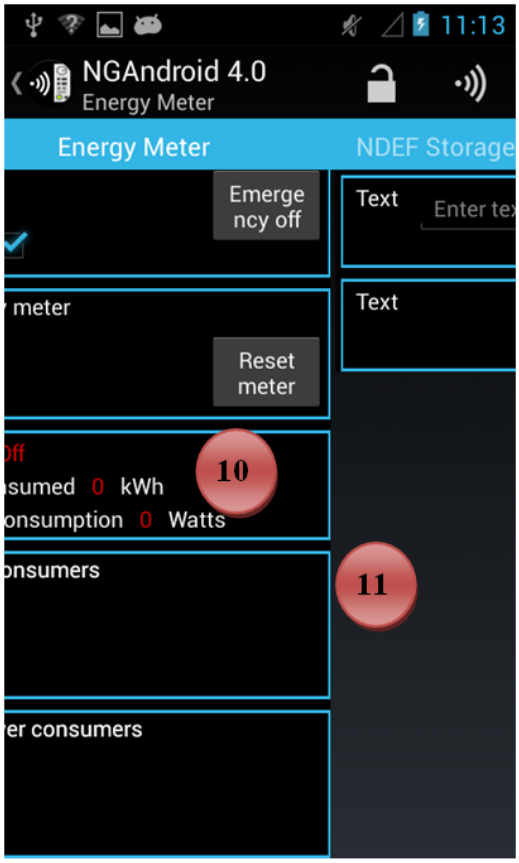

Figure 5.7: Android GUI showing swipe between nodes

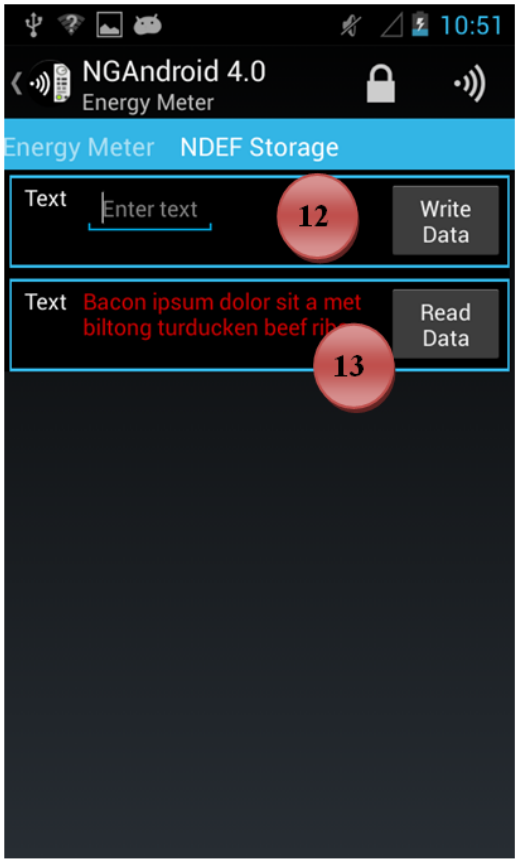

Figure 5.8: Android GUI allowing reading and writing of an NDEF message

Further functionality of the energy meter is depicted in Figures 5.7 and 5.8, including switching between two nodes and writing an NDEF message to the NFCI.

10. The energy meter has been reset by the user, power consumption has been reset to 0.

11. The user is currently swiping to the next node.

12. Any message can be written in this field in order to be stored on the Target Device as an NDEF message.

13. A dummy message has been stored before; the user has retrieved it via this element.

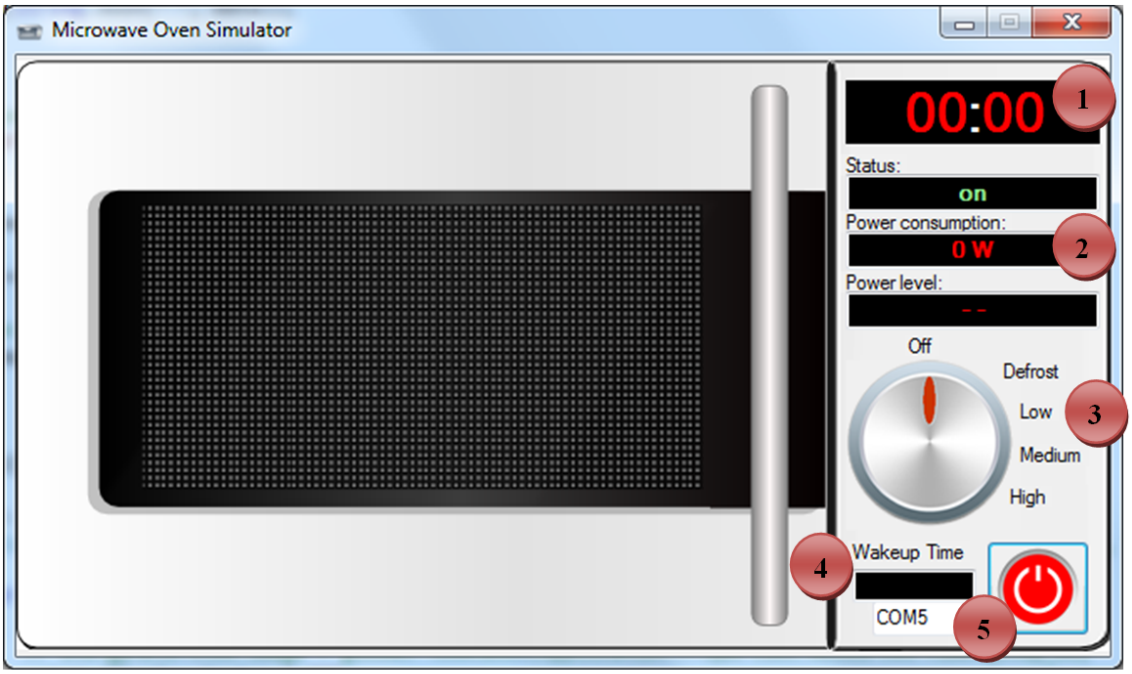

Figure 5.9: Microwave Oven PC application posing as a node attached to the NFC Interface (adapted from [Bas13])

5.5 Microwave Oven Use Case

The microwave oven use case allows a user to operate his microwave oven via his Android NFC-enabled smartphone. The Target Device (PC application) acts as the microwave oven, thus, it can cook at different power levels and keep track of the remaining time.

5.5.1 Target Device PC Application

Figure 5.9 depicts all functionality of the microwave oven node while in standby mode. Figure 5.10 shows the microwave while cooking. Their GUI components are now explained in more detail according to the enumeration in the Figures:

1. Shows the time remaining until the current program has finished.
2. These values show whether the device is switched on, as well as the current power consumption and the current power level.
3. A visual representation of the power level in the form of a knob.
4. The microwave can be configured to automatically stark cooking at a certain time via this control.
5. Shows the power button and the serial port used to connect the PC application to the NFC Interface.

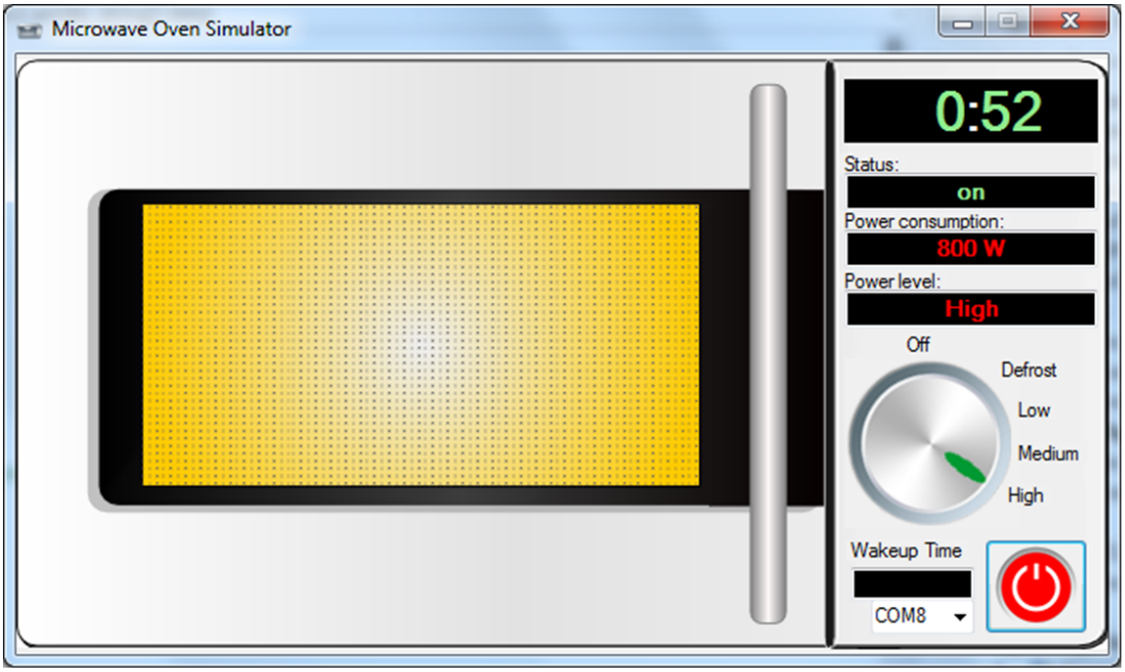

Figure 5.10: Microwave oven PC application simulating cooking at high power setting (adapted from [Bas13])

5.5.2 Android App

Figures 5.11 and 5.12 show the Android app GUI for the microwave oven while in standby and while running, including an enumeration of their functionality. The full TDD these Figures are based on can be found in Appendix A.2. An exact explanation of their GUI components is now given:

1. The user may choose an operation mode from a drop down list containing some usage suggestions (e.g. Boil water, Defrost, etc.).
2. The user may choose an operation mode by clicking on one of the images given. Each one represents an operation mode. The upper left image represents the “Boil water” mode, the upper right image the “Defrost” mode etc.
3. Shows the parameters available to configure the microwave oven’s operations. They will be set automatically when using the functionality described in 1 and 2, or may as well be modified and set manually by the user.
4. The microwave oven can be started and stopped by these buttons.
5. The microwave oven can be configured to start cooking automatically at a certain time.
6. The current status of the microwave oven can be monitored through this element.
7. The current status after the microwave oven has been activated is displayed.

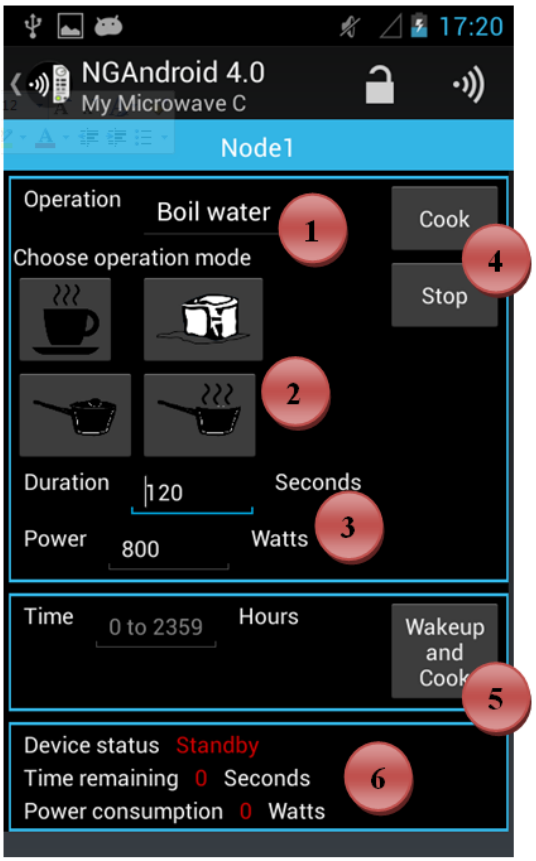

Figure 5.11: Android GUI for the microwave oven while in standby

Figure 5.12: Android GUI for the microwave oven while running

Chapter 6

Conclusions

This thesis focuses on improving the user experience and reducing integration complexity of the NFC Interface (NFCI) when incorporating it into household appliances. In order to achieve this, an XML-based descriptive markup language was introduced allowing the description of simple consumer electronic devices such as microwave ovens, washing machines or energy meters. The technology used for the language is based on what is being used in domains facing similar problems such as home automation. The Target Device Description (TDD) for a specific device can be stored on the NFCI hardware and may be downloaded at any time by a compatible reader.

An Android app able to download and use the TDD was developed as part of this thesis. The app can interpret the TDD and create a graphical user interface (GUI) based on it. The main goal of the GUI is to to ensure a user is offered the possibility to access the full functionality of the Target Device, while looking and feeling as if the Target Device is now a part of the smartphone itself. This has also been achieved by following the Android design guidelines.

The functionality of all parts of the system (NFCI, TDD and Android app) working together was practically tested as presented in chapter 5. Two specific use cases were implemented in order to demonstrate the full systems functionality: the microwave oven and the energy meter. Their respective functionality was implemented using PC applications acting as nodes connected to the NFCI via a USB connection and as such being part of the Target Device as a whole. Furthermore, a timing analysis was conducted to evaluate the transfer times for different TDDs via NFC.

6.1 Future Work

6.1.1 Security

Secure communication for the NFCI system has already been implemented by [Fio13]. The integration of this work into the one presented by this thesis is an important task.

6.1.2 Unified NDEF Message Based Communication Protocol

The current communication protocol for the NFCI system is split into two parts: TagType4-based communication for functionality belonging to the NFCI only and native command based communication for functionality belonging to nodes. The Target Device's firmware should be refactored so any command is being stored in an NDEF message on the NFCI before being relayed to the node by the firmware itself instead of directly routing the reader's communication there. This would introduce an important advantage by eliminating potential timeouts for commands taking a long time to be processed until a response is returned via the NFCI.

6.1.3 Resource Composition

Certain tags could contain information belonging to a certain type of Target Device, for example a recipe belonging to a microwave. Touching such a tag would then save the information stored on it and upon opening a compatible device's user interface within the NFCI app, the information is provided as an input suggestion amongst those provided by the TDD itself. This kind of collection system might additionally motivate a user to use the app as he may dynamically and intuitively customize the UIs functionality according to his preferences.

6.1.4 User Testing With Real Target Devices

A question remaining entirely open is whether users would be interested in controlling their devices via a smartphone instead of integrated hardware controls. Of course, in some cases it makes more immediate sense, like the energy meter that does not offer any hardware controls, or a radio alarm clock where a dozen or more buttons can be replaced by much clearer controls on the smartphone screen. Some use cases should be implemented in hardware in order to replace the current Target Device PC applications and a group of test users should be invoked in order to learn more about the acceptance of controlling devices via smartphones.

Appendix A

Target Device Descriptions

A.1 Energy Meter

Code A.1 XML Target Device Description of the Energy Meter

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <device>
3    <version>3</version>
4    <name>Energy Meter</name>
5    <description> Yet another energy meter! </description>
6    <node name="Energy Meter" id="1">
7      <controlElement name="Emergency off" type="Post" id="1">
8        <text>Turn off all power to one or more rooms</text>
9          <parameter name="Kitchen">
10           <id type="Immutable">0</id>
11           <unit textTrue="On" textFalse="Off">Boolean</unit>
12         </parameter>
13         <parameter name="Living Room">
14           <id type="Immutable">1</id>
15           <unit textTrue="On" textFalse="Off">Boolean</unit>
16         </parameter>
17     </controlElement>
18     <controlElement name="Reset meter" type="Secure Post" id="2">
19       <text>Reset the energy meter</text>
20     </controlElement>
21     <monitorElement name="Status" type="Output" id="3">
22       <parameter name="Device status">
23         <id type="Monitor">0</id>
24         <unit textTrue="On" textFalse="Off">Boolean</unit>
25       </parameter>
26       <parameter name="Total Power Consumed">
27         <id type="Monitor">1</id>
28         <unit>kWh</unit>
29       </parameter>
30       <parameter name="Current Power consumption">
31         <id type="Monitor">2</id>
32         <unit>Watts</unit>
33       </parameter>
34     </monitorElement>
35     <monitorElement name="Kitchen" type="Output" id="4">
```

```
36      <text>Kitchen power consumers</text>
37        <parameter name="Stove">
38          <id type="Monitor">0</id>
39          <unit textTrue="On" textFalse="Off">Boolean</unit>
40        </parameter>
41        <parameter name="Oven">
42          <id type="Monitor">1</id>
43          <unit textTrue="On" textFalse="Off">Boolean</unit>
44        </parameter>
45        <parameter name="Light">
46          <id type="Monitor">2</id>
47          <unit textTrue="On" textFalse="Off">Boolean</unit>
48        </parameter>
49      </monitorElement>
50      <monitorElement name="Living Room" type="Output" id="5">
51      <text>Living room power consumers</text>
52        <parameter name="TV">
53          <id type="Monitor">0</id>
54          <unit textTrue="On" textFalse="Off">Boolean</unit>
55        </parameter>
56        <parameter name="AVR">
57          <id type="Monitor">1</id>
58          <unit textTrue="On" textFalse="Off">Boolean</unit>
59        </parameter>
60        <parameter name="Light">
61          <id type="Monitor">2</id>
62          <unit textTrue="On" textFalse="Off">Boolean</unit>
63        </parameter>
64      </monitorElement>
65    </node>
66    <node name="NDEF Storage" id="2">
67      <controlElement name="Write Data" type="WriteNDEF" id="1">
68        <nDEFMessage>E106</nDEFMessage>
69        <parameter name="Text">
70        <id type="Immutable">0</id>
71        <unit>Text</unit>
72        </parameter>
73      </controlElement>
74      <configurationElement name="Read Data" type="ReadNDEF" id="2">
75        <nDEFMessage>E106</nDEFMessage>
76        <parameter name="Text">
77        <id type="Monitor">0</id>
78        <unit>Text</unit>
79        </parameter>
80      </configurationElement>
81    </node>
82  </device>
```

A.2 Microwave Oven

Code A.2 XML Target Device Description of the Microwave oven

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<device>
  <version>3</version>
  <name>My Microwave B</name>
  <description> Yet another microwave! </description>
  <node name="Node1" id="1">
    <controlElement name="Cook" type="Post" id="1">
      <parameter name="Duration">
        <id type="Immutable">0</id>
        <unit minValue="1" maxValue="1800">Seconds</unit>
      </parameter>
      <parameter name="Power">
        <id type="Immutable">1</id>
        <unit minValue="10" maxValue="800">Watts</unit>
      </parameter>
      <parameter name="Device status">
        <id type="Immutable">2</id>
        <unit>Boolean</unit>
        <visibility>invisible</visibility>
      </parameter>
      <nestedControlElement name="Stop" type="Post" id="2">
        <parameter>None</parameter>
      </nestedControlElement>
      <userInputAid name="Choose operation mode" type="Grid">
        <icon Duration="120" Power="800">
          <name imgsrc="boil.png">Boil water</name>
        </icon>
        <icon Duration="480" Power="80">
          <name imgsrc="defrost.png">Defrost</name>
        </icon>
        <icon Duration="240" Power="400">
          <name imgsrc="heat.png">Heat food</name>
        </icon>
        <icon Duration="240" Power="800">
          <name imgsrc="cook.png">Cook food</name>
        </icon>
      </userInputAid>
    </controlElement>
    <configurationElement name="Wakeup and Cook" type="Post" id="3">
      <combineElements>false</combineElements>
      <parameter name="Time">
        <id type="Configuration">0</id>
        <unit minValue="0" maxValue="2359">Hours</unit>
      </parameter>
      <parameter name="Device status">
        <id type="Monitor">1</id>
        <unit>Boolean</unit>
        <visibility>invisible</visibility>
      </parameter>
    </configurationElement>
    <monitorElement name="Status" type="Output" id="4">
      <parameter name="Device status">
```

```
53          <id type="Monitor">0</id>
54          <unit textTrue="Active" textFalse="Standby">Boolean</unit>
55        </parameter>
56        <parameter name="Time remaining">
57          <id type="Monitor">1</id>
58          <unit>Seconds</unit>
59        </parameter>
60        <parameter name="Power consumption">
61          <id type="Monitor">2</id>
62          <unit>Watts</unit>
63        </parameter>
64      </monitorElement>
65    </node>
66  </device>
```

Appendix B

System Setup

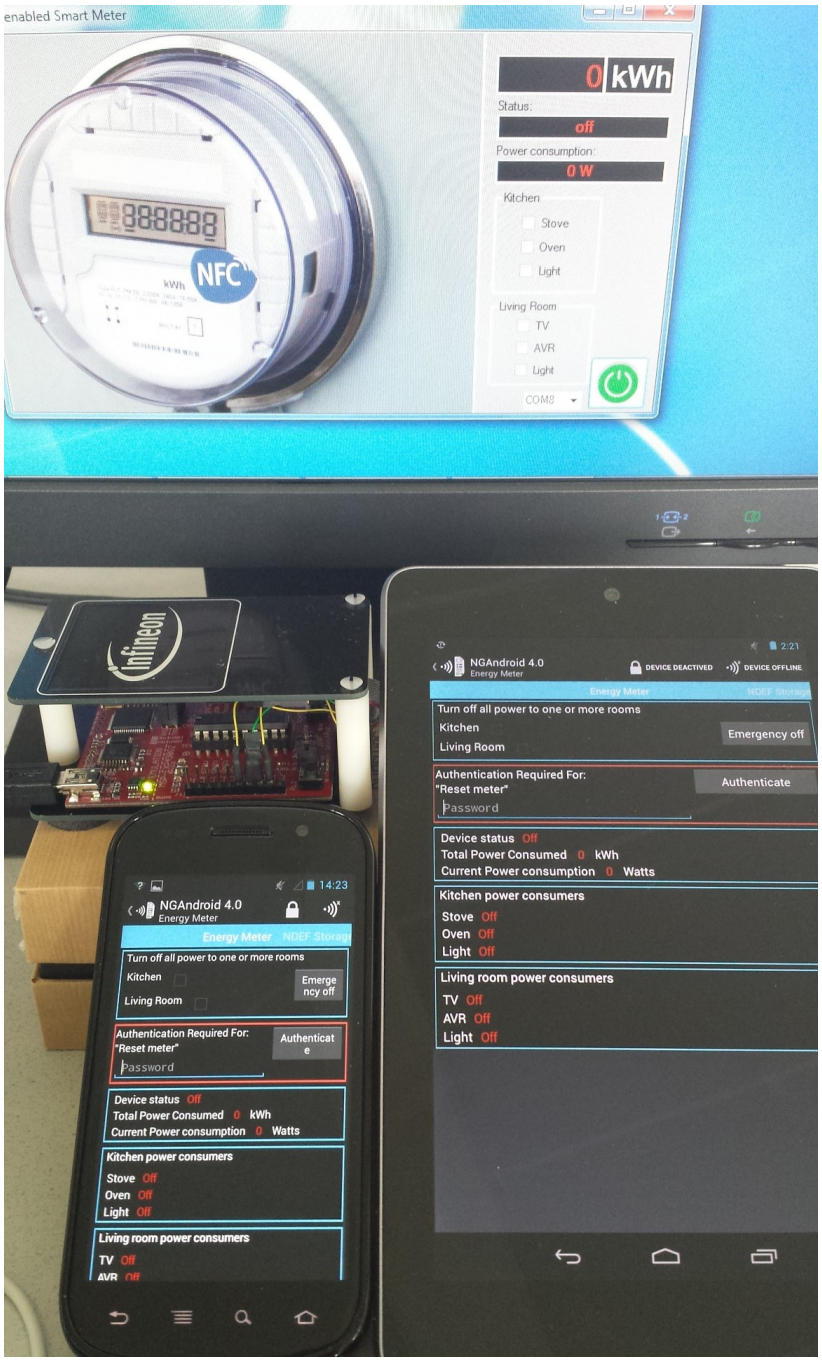

Figure B.1: Photograph of the complete NFC Interface system setup

Glossary

- **Actuator** - A hardware device which represents an entity only taking input, however, not producing any output (e.g., a light switch).
- **Android SDK** - Android System Development Kit
- **AES** - Advanced Encryption Standard
- **Device** - A hardware device taking input and producing output.
- **dp** - density-independent pixels; an abstract unit based on the physical density of a screen [1]
- **ECC** - Elliptic Curve Cryptography
- **GUI** - Graphical User Interface
- **NFC** - Near Field Communication[2]; a technology used to exchange data over very small distances via radio waves. Must not be confused with RFID. NFC is a subset of RFID.
- **NFCI** - Near Field Communication Interface (see chapter 1.2)
- **RFID** - Radio Frequency Identification; a popular technology widely used in many industries (e.g. for tracking of parts in the automobile industry). RFID tags often use UHF radio waves for communication over longer distances (e.g. 1 meter). UHF is not supported by NFC and most RFID tags are not NFC tags. The two must not be confused.
- **Sensor** - A hardware device typically used to gather specific real-world data, such as temperature, humidity or brightness. It is characterized as an entity that only provides output data without taking any input data.
- **Smartphone** – A handheld computer device which can be used as a phone, however offers more functionality than that. Smartphones relevant to this project have to support NFC technology.
- **Tablet** - Has a larger screen than a smartphone while offering similar functionality. Operating system and apps might be specifically optimized towards it.

[1] http://developer.android.com/guide/topics/resources/more-resources.html#Dimension
[2] http://www.nfc-forum.org/home/

- **TDD** - Target Device Description. The description of a Target Device's functionality via the XML based, domain specific descriptive markup language implemented in this thesis.
- **Transducer** - see **Device**
- **UART** - Universal Asynchronous Receiver Transmitter
- **UI** - User Interface
- **XML** - Extensible Markup Language

Bibliography

[AV07] Z. Antoniou and S. Varadan. Intuitive Mobile User Interaction in Smart Spaces via NFC-Enhanced Devices. In *Wireless and Mobile Communications, 2007. ICWMC '07. Third International Conference on*, pages 86–86, 2007.

[Bas13] Rejhan Basagic. Design and Implementation of an NFC Interface for Home Appliances and Consumer Electronics. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2013.

[BHP+08] Gregor Broll, Markus Haarlaender, Massimo Paolucci, Matthias Wagner, Enrico Rukzio, and Albrecht Schmidt. Collect&Drop: A Technique for Multi-Tag Interaction with Real World Objects and Information. In Emile Aarts, James L. Crowley, Boris de Ruyter, Heinz Gerhäuser, Alexander Pflaum, Janina Schmidt, and Reiner Wichert, editors, *Ambient Intelligence*, Lecture Notes in Computer Science, pages 175–191. Springer, 2008.

[CH08] Chao Chen and S. Helal. Sifting Through the Jungle of Sensor Standards. *Pervasive Computing, IEEE*, 7(4):84–88, 2008.

[CPL11] Longbiao Chen, Gang Pan, and Shijian Li. Touch-Driven Interaction between Physical Space and Cyberspace with NFC. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 258–265, 2011.

[CPL12] Longbiao Chen, Gang Pan, and Shijian Li. Touch-driven interaction via an NFC-enabled smartphone. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference*, pages 504–506, 2012.

[DM12] Norbert Druml and Manuel Menghin. "NIZE - A Near Field Communication Interface Enabling Zero Energy Standby for Everyday Electronic Devices". *RSTR-002-97-002.01, Reliable Software Technologies Corporation*, 2012.

[Fio13] Manuel Trebo Fioriello. Trusted Device Interaction over NFC. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2013.

[For06a] NFC Forum. NFC Data Exchange Format (NDEF), Technical Specification 1.0. Technical report, 2006.

[For06b] NFC Forum. NFC Record Type Definition (RTD), 1.0. Technical report, 2006.

[For08] NFC Forum. NFC-Forum: Essentials for Successful NFC Mobile Ecosystems. Technical report, 2008.

[For11] NFC Forum. Type 4 Tag Operation Specification, Technical Specification, T4TOP 2.0. Technical report, 2011.

[GR11] M. Ganchev and U. Reisenbichler. Design of a general purpose user interface for realtime embeded systems. In *EUROCON - International Conference on Computer as a Tool (EUROCON), 2011 IEEE*, pages 1–4, 2011.

[HB06] Ernst Haselsteiner and Klemens Breitfuss. Security in Near Field Communication. 2006.

[HiYKR07] Sumi Helal, Hen i Yang, Jeffrey King, and Raja. Atlas- Architecture for Sensor Network Based Intelligent Environments. `http://www.cise.ufl.edu/~hyang/doc/AtlasArchitecture_TOSN_0224.pdf`, 2007. Accessed: 2013-05-12.

[Huf03] M. F. L Hufkens. XML for embedded systems. Department of Mathematics and Computer Science, 2003.

[IEE07] IEEE. IEEE Standard for a Smart Transducer Interface for Sensors and Actuators - Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats. Technical report, 2007.

[IEE10] IEEE. IEEE Standard for Smart Transducer Interface for Sensors and Actuators - Transducers to Radio Frequency Identification (RFID) Systems Communication Protocols and Transducer Electronic Data Sheet Formats. Technical report, 2010.

[ISO05] ISO. ISO/IEC 7816 Part 4: Identification cards - Cards with contacts Organization, security and commands for interchange. Technical report, 2005.

[Jon05] R. Jones. XML provides extra help for embedded systems. *Electronics Systems and Software*, 3(5):26–31, 2005.

[LC01] Kris Luyten and Karin Coninx. An XML-based runtime user interface description language for mobile computing devices, 2001.

[MPCL08] MOBILE and UNIVERSITY OF FLORIDA PERVASIVE COMPUTING LABORATORY. Device Description Language Specification Version 1.2. Technical report, 2008.

[OEB] OEBB. OEBB Innovationstag. `http://konzern.oebb.at/de/Presse/Presseinformationen_aus_den_Bundeslaendern/Niederoesterreich/PDF/2012/Q2/2012_06_20_PI_Innovationstag_Gewinnspiel.pdf`. Accessed: 2013-06-20.

[Ogb04] Uche Ogbuji. Principles of XML design: When to use elements versus attributes. `http://www.ibm.com/developerworks/library/x-eleatt/index.html`, 2004. Accessed: 2013-08-21.

[Pae11] Christian Paetz. *Z-Wave Grundlagen: Funksteuerung im Smart Home*. Books on Demand, 2011.

[Pha00] Constantinos Phanouriou. UIML: A Device-Independent User Inter-face Markup Language. Virginia Polytechnic Institute and State University. 2000.

[RG] Rohde&Schwarz-GmbH. Near Field Communication (NFC) Technology and Measurements White Paper. Accessed: 2013-06-20.

[SRP09] Iván Sánchez, Jukka Riekki, and Mikko Pyykkönen. Touch & Compose: Physical User Interface for Application Composition in Smart Environments. In *Proceedings of the 2009 First International Workshop on Near Field Communication*, NFC '09, pages 61–66, Washington, DC, USA, 2009. IEEE Computer Society.

[Tor08] R. Torbensen. OHAS: Open home automation system. In *Consumer Electronics, 2008. ISCE 2008. IEEE International Symposium on*, pages 1–4, 2008.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American*, 265:94–104, 1991.