Design of a Flexible UHF RFID TAG with Security Functionality Based on an 8-Bit Microcontroller

Krassimir Duschkov
duschkov@student.tugraz.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master's Thesis

Assessor: Ao. Univ.-Prof. Dipl.Ing. Dr.techn. Karl-Christian Posch
Supervisor: Dipl.Ing. Dr.techn. Thomas Plos,
Dipl.Ing. Dr.techn. Erich Wenger

Mai, 2015

I hereby certify that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by reference to other authors.

Ich bestätige hiermit, diese Arbeit selbständig verfasst zu haben. Teile der Diplomarbeit, die auf Arbeiten anderer Autoren beruhen, sind durch Angabe der entsprechenden Referenz gekennzeichnet.

Krassimir Duschkov

Acknowledgements

First of all, I would like to thank Thomas Plos and Erich Wenger. They always took time to give me support and advice. Thank you for your supervision and patience during the progress of my work.

Secondly I would like to thank my girlfriend and my family. They kept encouraging me and gave me the mental support when I needed it. I also would like to thank my employer Johann Sauseng-Weiss for his sincere appreciation for my education.

Finally I would also like to thank Alex Apostolov for proofreading my thesis.

Abstract

The electronic product code (EPC) is seen as a successor of the bar-code. Together with the RFID technology it is expected that for the EPC a broad field of applications will arise in the future. Thus a flexible design of an EPC tag with the possibility of adapting the functionality in software is desirable.

RFID technologies bring huge advantages in many applications of people's everyday lives and countless future applications are imaginable. Involving people's everyday lives with an application based on RFID, like the EPC, brings up reasonable security concerns on such RFID devices. Privacy and authenticity are the two main issues mentioned in the context of RFID security. Therefore security functionalities play a large role in different systems based on RFID.

The first version of the EPC Gen2 standard does not provide any security functionality. An EPC tag based on a flexible design that is adaptable for the security issues in a certain application would save on many costs for the assimilation of given security features.

This thesis presents a design based on an 8-bit microcontroller that implements the AVR instruction set of the Atmega128. It has a front-end extension that handles the fundamental functionality for communication according to the EPC Gen2 standard.

The processing of the tag's incoming and outgoing data is split to be handled by the front-end as well as by the software on the microcontroller. The software for the handling of the basic commands is written in the programming language C, whereas the critical parts of the software are highly optimized in assembly. This allows one to clock the microcontroller with a minimal cycle count to execute the command handling. Since low power is one of the mandatory requirements for RFID devices, a low clock frequency is important for reducing the power consumption.

Further, the front-end is able to clock the microcontroller dynamically, depending on the current communication process and data rate. Thus, not only is the required cycle count reduced, but also the clock frequency can be adapted to the data rate.

In an RFID application the power that is supplied to the tag by a reader depends on the distance between tag and reader. With larger distance less power is supplied to the tag. With a power consumption that depends on the data rate, it is possible to deliberate between the communications distance and the data rate.

In this design the minimal effective clock frequencies are 186.7 kHz for reception and 25 kHz for transmission, assuming the lowest possible data rates, whereas for the highest possible data rates, the clock frequency results in 896 kHz for reception and 3200 kHz for transmission.

Further, the handling of additional security features and commands can be implemented easily in software with the tag presented in this thesis. The design of the tag requires only adaption of the command handling in the software in the programming language C.

Using the resulting hardware design, a simple authentication scheme is implemented with a software extension for additional security features allowing the authentication of the tag to a reader. The conformance of the final design including the authentication functionality has been verified on a demonstration tag with a Field Programmable Gate Array (FPGA).

Keywords: Radio Frequency Identification, EPC Gen2 tag, flexible design, standard microcontroller, security functionality

Kurzfassung

Der Electronic Product Code (EPC) wird als Nachfolger des Barcodes angesehen. Zusammen mit der Radio Frequency Identification (RFID) Technologie ist zu erwarten, dass für den EPC zukünftig ein breites Feld an Anwendungen entstehen wird. Hierfür ist ein flexibles Design eines EPC Tags, mit der Möglichkeit die Funktionalität durch Software zu adaptieren, vorteilhaft.

Radio-Frequency Identification (RFID) Technologien bringen nicht nur große Vorteile in vielen Anwendungen des alltäglichen Lebens, sondern es sind damit auch zahllose zukünftige Anwendungen vorstellbar. Das Involvieren des alltäglichen Lebens mit Anwendungen basierend auf RFID, wie der Electronic Product Code (EPC), wirft ernsthafte Bedenken an die Sicherheit von RFID Applikationen auf. Vertraulichkeit (englisch: privacy) und Authentizität (englisch: authenticity) sind zwei der Hauptaspekte, welche in diesem Zusammenhang genannt werden. Aus diesem Grund spielen in den verschiedenen Systemen, die auf RFID basieren, Sicherheitsfunktionen eine große Rolle.

Die erste Version des EPC Gen2 Standards bietet keine Sicherheitsfunktionen. Ein EPC Tag, basierend auf einem flexiblen Design, welches in Punkto Sicherheit für die jeweilige Anwendung adaptierbar ist, würde eine Menge Kosten für die Anpassung der Sicherheitsfunktionen sparen.

Diese Arbeit präsentiert ein Design basierend auf einem 8-Bit Mikrocontroller, welcher das Atmega 128 Instruction Set implementiert. Es besitzt ein Front-End, das die grundlegenden Aufgaben der Kommunikation hinsichtlich des EPC Gen2 Standards erledigt.

Die Verarbeitung der empfangenen und gesendeten Daten eines Tags ist sowohl auf das Front-End, als auch auf die Software des Mikrocontrollers aufgeteilt. Die Software zur Handhabung der grundsätzlichen Kommandos ist in der Programmiersprache C geschrieben, wohingegen die kritischen Teile der Software in Assembly optimiert sind. Dadurch ist es möglich den Mikrocontroller mit einer minimalen Anzahl an Zyklen zu takten. Da für RFID Anwendungen ein niedriger Energieverbrauch entscheidend ist, ist eine minimale Taktfrequenz ausschlaggebend, um den Energieverbrauch zu reduzieren.

Das Front-End ist in der Lage den Mikrocontroller dynamisch zu takten, abhängig von dem aktuellen Kommunikationsfluss und der Datenrate. Dadurch wird nicht nur die Anzahl der benötigten Taktzyklen reduziert, sondern es kann auch die Taktfrequenz an die Datenrate angepasst werden.

In einer RFID Anwendung ist die Energie, welche durch das Lesegerät an das Tag übertragen wird, abhängig vom Abstand zwischen dem Tag und dem Lesegerät. Mit größerem Abstand wird das Tag mit weniger Energie versorgt. Mit einem Energieverbrauch der abhängig ist von der Datenrate, ist es möglich zwischen Kommunikationsabstand und Datenrate abzuwägen.

In diesem Design beträgt die minimale effektive Taktfrequenz des Tags für die niedrigste Datenrate 186.7 kHz während des Empfangens und 25 kHz während des Sendens,

wohingegen für die höchste Datenrate eine Taktfrequenz von 896 kHz zum Empfangen und 3200 kHz zum Senden benötigt wird.

Ferner kann, mit dem in dieser Arbeit vorgestellten Tag, die Verarbeitung von zusätzlichen Sicherheitsfunktionen und Kommandos einfach in Software implementiert werden. Aufgrund des Designs des Tags, ist lediglich eine Anpassung der Software in der Programmiersprache C für die Verarbeitung von zusätzlichen Kommandos notwendig.

Im endgültigen Design wurde durch einfache Softwareerweiterung eine Funktion zur Authentifizierung implementiert, welche es ermöglicht, dass ein Tag durch ein Lesegerät authentifiziert wird. Die Korrektheit der Funktion des letztendlichen Designs inklusive der Authentifizierungsfunktion konnte an einem Demonstrationstag mit einem Field Programmable Gate Array (FPGA) verifiziert werden.

Stichwörter: Radio Frequency Identification, EPC Gen2 Tag, Flexibles Design, Standard Mikrocontroller, Sicherheits Funktionen

Contents

1 Introduction **1**

2 Introduction to RFID **4**
- 2.1 A Brief History of RFID Technology 4
- 2.2 RFID Applications 5
- 2.3 Security and Privacy Challenges of Tags 7
  - 2.3.1 Privacy 7
  - 2.3.2 Authenticity 8
- 2.4 Security-Efforts on EPC-Tags 9

3 Security in Communication **11**
- 3.1 Cryptographic tools 11
  - 3.1.1 Symmetric Crypto Primitives 13
    - 3.1.1.1 Block Ciphers 13
    - 3.1.1.2 Stream Ciphers 15
  - 3.1.2 Asymmetric Crypto Primitives 16
  - 3.1.3 Hash Functions 17
- 3.2 Advanced Encryption Standard 18
- 3.3 Protocols 22

4 The EPC Gen2 standard **24**
- 4.1 Signaling 24
  - 4.1.1 Communication from Reader to Tag 24
  - 4.1.2 Communication from Tag to Reader 25
  - 4.1.3 Link Timing 27
- 4.2 Mechanisms for Tag Access 28
  - 4.2.1 Tag Memory 28
  - 4.2.2 Inventoried Flag and Selected Flag 29
  - 4.2.3 States of a Tag 29
  - 4.2.4 Managing Tags 31
    - 4.2.4.1 Selecting Step 31
    - 4.2.4.2 Inventorying Step 31
    - 4.2.4.3 Accessing Step 32
- 4.3 Possible Extensions and Improvements 33

5 Architecture **34**
5.1 System Overview . . . 34
5.2 Trade-off between Hardware and Software . . . 35
5.3 Design of the Front-end . . . 37
5.3.1 Rx-Tx-module . . . 37
5.3.2 FIFO . . . 39
5.3.3 Pseudo Random Number Generator . . . 40
5.4 Clock Frequency Constraints . . . 40
5.5 Optimization Approaches in Software . . . 42
5.6 Control flow between Hardware and Software . . . 44
5.6.1 Controlling the Front-end . . . 45
5.6.2 Sensing the front-end . . . 45
5.6.3 Critical points of operation . . . 46

6 Implementation and Evaluation **48**
6.1 Simulated Model of the System . . . 49
6.2 Test Case Generation and Appliance . . . 49
6.3 Extending the Protocol . . . 50
6.4 Evaluation of the Software . . . 51
6.4.1 Optimization Results . . . 51
6.4.2 Clock Requirement . . . 52

7 Concluding Remarks **58**

A Definitions **60**
A.1 Acronyms . . . 60

B Interface Register **62**
B.1 Status 1 Register . . . 62
B.2 Status 2 Register . . . 63
B.3 Control 1 Register . . . 64
B.4 Control 2 Register . . . 65

C Algorithms **67**
C.1 Extended Euclidean algorithm . . . 67

Bibliography **70**

List of Figures

2.1 Tracking a Person 8
2.2 Tag being spied and copied 9

3.1 Two parties using an encryption scheme 12
3.2 Substitution-permutation network 13
3.3 Electronic Code Book 14
3.4 Cipher Block Chaining 14
3.5 Output Feedback 15
3.6 Synchronous Stream Cipher 15
3.7 Feedback Shift Registers 16
3.8 Linear Feedback Shift Registers 16
3.9 *SubBytes* Transformations 19
3.10 *ShiftRows* Transformations 19
3.11 MixColumns Transformations 20
3.12 AddRoundKey Transformations 21
3.13 Types of Protocols 22
3.14 Types of Protocols 23
3.15 Types of Protocols 23

4.1 PIE Symbols 25
4.2 Interrogator to tag preamble and frame-sync 25
4.3 FM0 Signals 26
4.4 Miller Signals 26
4.5 FM0 Preamble 27
4.6 Miller Preamble 27
4.7 Link timing 27
4.8 Actual T1 link timing 29
4.9 Memory Distribution 30

5.1 System overview 34
5.2 System overview 38
5.3 System overview 39
5.4 Pseudo Random Number Generator 40
5.5 Power distribution 41
5.6 Control flow without delay 46
5.7 Control flow with delay 47

6.1 Evaluation of minimal RX-BIT-RATE requirement 56

List of Tables

4.1 Link timing 28

5.1 Special Command-Detection 39
5.2 Feedback polynomials for the PRNG 40

6.1 Comparison of *Select* command optimization steps 51
6.2 Comparison of *ACK* command optimization steps 52
6.3 Dependency between TX-WAIT-CYCLES and the RX-BIT-RATE for the *Select* command 53
6.4 Dependency between TX-WAIT-CYCLES and the RX-BIT-RATE for the *Query* command 54
6.5 Required TX-WAIT-CYCLES for the *QueryAdjust* command 55
6.6 Required TX-WAIT-CYCLES for the *QueryRep* command 55
6.7 Effective Clock-rate for receiving 56
6.8 Effective Clock-rate for transmitting 57

B.1 Status_1 Register 62
B.2 Status_2 Register 63
B.3 Control_1 Register 64
B.4 Control_2 Register 65

Chapter 1

Introduction

The use of RFID reaches back to the 50's of the last century. Using RFID technology an object can be uniquely identified through wireless transmission of its identity. An RFID system basically consists of a tag, which stores the identity of an object, and an interrogator, that reads the identity through an electromagnetic field. During the process of communication, the electromagnetic field, transmitted by the interrogator, can also be used to power the tag.

In many applications like supply-chain management, inventory control and logistics RFID technologies can help to operate more flexibly and save labor and time (see Weis [1] and Roberti [2]). But also in other applications like toll collection (see Dong-Liang et al. [3]), access control (see Weis [1]), contactless payment (see Lacmanovic et al. [4]) or animal identification (see Jeffries [5]) the use of RFID technology brings many benefits. Nevertheless, involving RFID in so many aspects of everyday life is of concern to many. The most frequently mentioned security concerns in the context of RFID are privacy and authenticity. The security risk within the RFID technology and eventual counter measurements are mentioned in Juels [6] and Pateriya and Sharma [7].

The main privacy issues are clandestine tracking and inventorying. Since in many applications, an RFID tag is powered wirelessy over the air, an individual has no control over the activity of its tags. A tag attached to an object that is worn by a person can be tracked and eventually linked to the identity of a certain person. Additionally, sensible data stored on an RFID device is a privacy issue (e.g., an E-passport mentioned in see Bogari et al. [8]). Authentication issues on the other hand have concerns regarding the information that is read from a tag. For tags that are used as proof of origin or for access control it should not be possible to counterfeit them by eavesdropping on the communication process. Thus modern RFID devices offer minor security functionalities for the certain application (see Dong-Liang et al. [3] and Juels [6]).

The EPC is an international system for identifying physical objects. Together with RFID technology it is very likely that EPC tags are going to be a successor of bar-codes. Further, it is expected that the applications of EPC tags will also spread into a broader field than it already has. Moreover the different applications will require different security functionalities (see Juels [6]). Thus it is hard to imagine one standard tag that can meet all the security requirements arising with different scopes of applications. Likewise, in the past a suggested security mechanism was erroneous and thus had to be modified, adapted or redesigned (e.g., Bagheri et al. [9] and Song et al. [10]).

Adapting the design of a tag for a certain application or modifying the security mech-

anism can have high costs. Since most of the designs have security features that are implemented in hardware, the functionality of such tags can not be updated and reused. For these reasons having a tag with a dedicated hardware part, targeted for the EPC Gen2 standard, but with a higher level functionality running on a microcontroller implemented in software would be an advantage for eventual updates of the functionality. Also it would be possible to have one hardware design that can be reused for different applications, while only the software has to be adapted for a new functionality.

This thesis presents a hardware design that is based on a microcontroller that implements a commonly used instruction set, the instruction set of an ATmega128. The microcontroller is extended by a front-end that is responsible for the communication according to the EPC Gen2 standard (see EPCglobal Inc. [11]). Further, this design splits the processing of the incoming and outgoing messages in a hardware part, handled by the front-end, and a software part, handled by the microcontroller. The front-end handles time critical operations and processes the data insofar as the microcontroller can handle the higher level functionality with an appropriate amount of cycles. Processing the data only in software would require a lot more cycles compared to the split design in this thesis. The communication protocol according to the EPC Gen2 standard requires certain timing constraints to be meet, a higher amount of required cycles would lead to a higher clock frequency.

A low clock frequency is one of the keys to reducing the power consumption of a device. Applications using RFID technology are typical low power applications, thus a low clock frequency is one of the goals in this thesis. To reduce the required cycles for the software handling, the time critical parts are highly optimized in assembly, whereas the less complex parts are implemented in the programming language C.

Further the clock frequency that is gated to the microcontroller is adjusted by the front-end, depending on the data rate of the current reader. Thus with a lower data rate the required cycles for the command handling are split over a larger time slot which reduces the clock frequency and further the power consumption that is needed to meet the timing constraints. Since the power that is supplied by a reader reduces the larger the distance gets between the tag and the reader, it is possible to achieve larger communications distances by reducing the data rate of the communication.

A software model is implemented that reflects the intended hardware design. On the one hand it is used to evaluate the interface and the data flow between the microcontroller and the front-end. On the other hand it is used to evaluate the software for the command handling on the microcontroller.

The following work is structured as follows:

An introduction into RFID and its history is given in Chapter 2. Possible RFID applications and the concerns of the security of RFID tags are briefly discussed. Moreover, existing efforts are given on security related to EPC tags.

Chapter 3 gives an introduction into security and cryptography. The basic cryptographic primitives are discussed and some examples are demonstrated.

Since the design of the presented implementation builds on the EPC Gen2 standard, the standard is summarized in Chapter 4. Critical characteristics that are also important for the thesis are pointed out.

In Chapter 5 an overview of the concept and the architecture of the system is given. Further, the trade-offs are discussed between hardware and software for implementation of the EPC Gen2 standard respectively regarding the cycle count requirement. Optimizations in software and the effects on the control flow between the front-end and software on the

microcontroller are analyzed.

Chapter 6 presents the implementation of a software model for verifying the concept. The software and the control flow between the software and the front-end are evaluated with the model. The software that runs on the microcontroller and implements the command handling according to the EPC Gen2 standard is extended by a custom command with a security functionality. Conclusions and remarks for further work are given in Chapter 7.

Chapter 2

Introduction to RFID

Radio frequency identification (RFID) is a technology, used for wireless identification of objects and entities. The system consists of a reader (or interrogator) and a transponder (or tag). The interrogator sends and reads information from the tag by means of electromagnetic waves. Passive tags are simultaneously supplied with energy by those electromagnetic waves, during the process of communication.

2.1 A Brief History of RFID Technology

In the 50s and 60s of the 20th century the first predecessors of modern RFID systems where developed. Actually, during World War II the Identify Friend or Foe (IFF) system was already used to identify an incoming airplane by radar. For this purpose planes had mounted special 'reflectors', which backscattered a certain signal. Thus on the radar they were identified as friendly forces.

In the late 60s Electronic Article Surveillance (EAS) tags were realized as 1-bit tags. In fact, only the presence or absence of the tag was detected. Such tags were used as an anti-theft measure in merchandise. The cashier in a store was able to turn the tag off. If a tag attached to a piece of merchandise was still turned on, this denoted that the item was not paid for.

With the development of the transistor and integrated circuits in the 70s and 80s, the tags became smaller and the functionality of the system more complex. Toll systems with tags in vehicles and readers at certain points on roads, bridges or tunnels were established in many countries. The correspondent tags were mounted on the cars and registered in a centralized database system. Thus, it was possible to automatically verify which parts of a road network were passed by a registered vehicle.

The first more contemporary low power applications with RFID tags were tags that were used as implants to identify animals. Such tags were implemented as passive Low-Frequency (LF) tags with a transponder frequency set at 125kHz. They were used on livestock on farms, so that a farmer could not run the risk of giving an animal an overdose by administering medicine twice. Additionally, tags at large meadows shared amongst different farmers could allow easy separation of cattle by owner.

In the late 80s and 90s, passive High Frequency (HF) tags, operating with a carrier of 13.56 MHz, offered a longer operating range than the LF tags. Such tags have been used in warehouse logistic tracking for goods and also in smart card systems. With the development of personal computers (PC), the data amounts resulting from large RFID

systems became manageable in a convenient and economical way, which expedite the expansion of RFID applications (see Landt [12]).

The first Ultra High Frequency (UHF) RFID systems were developed in the 90s, but in the beginning they were to expensive to get proper acceptance. With the foundation of the Auto-ID Center at the Massachusetts Institute of Technology the research on UHF systems increased. The Auto-ID Center developed the electronic product code (EPC) standard, an universal identifying scheme. This led to founding the EPCglobal Inc in 2003. Since 2004 the EPC Gen2 standard has been developing, and is nowadays the standard for the most quickly growing field involving RFID technology.

For more information about the history of RFID technology see Landt [12], Violino [13] and Dong-Liang et al. [3].

2.2 RFID Applications

RFID systems can be distinguished based on the transponder frequencies used:

- **Low Frequency** systems work in the frequency spectrum of 120-140 kHz which is in general an unregulated spectrum. The tags are mainly passively powered and have a short operating range (10-20cm). Such systems offer only a low data rate in comparison to other RFID systems. One major advantage is that they could be used in a rugged environment like metal, liquid or dirt. So one typical application for such tags is animal identification where glass encapsulated tags are implanted into pets (see Weis [1]).

- The **High Frequency** spectrum is the range from 3 MHz to 30 MHz. This is a strongly regulated part of the frequency spectrum. RFID systems in the HF range work at a frequency of 13.56 MHz which is specified in the ISO 18000-3 standard (see ISO/IEC 18000-3 [14]). HF tags have a higher data rate and a greater operating range (up to one meter for passive tags) than LF tags systems. Typically such devices are implemented as passive tags and printed onto a foil or packed into a plastic card used for access control (see Weis [1]).

- **Ultra High Frequency** tags work at a frequency spectrum of 860-960MHz. In the United States and Canada (902-928MHz) a different frequency spectrum is specified by the ISO 18000-6 standard (see ISO/IEC 18000-6 [15]) compared to European countries (865-870MHz). UHF tags have a higher data rate and a longer operating range than HF tags (three to six meters for passive tags). Such systems are mainly used in supply chain management (see Weis [1]). This is because of their long operating range, but also because of the low price per unit (see Weis [1]).

Today, various RFID systems are implemented in a lot of different applications. Only a few of them shall be mentioned as representatives of the most common applications:

- **Tracking of Goods**
  Supply chain management is one of the largest application areas and a good example for large systems based on RFID. In this application, the tags are mounted on the items that shall be tracked throughout the supply chain. Knowing the stage of a certain item during a supply chain helps with organizing the massive amounts

of goods in supply chain management. The use of transponders on containers and different points of active control events can support the complex material flow. Because of this, the logistics management can operate more flexibly, which saves labor and time. Finally, this reduces the costs markedly (see Weis [1]). In retail RFID technology, more precisely the EPC, seems to replace the technology of the barcode. In comparison to the barcode, where it is only possible to scan a single item at a time, using RFID technology one reader can scan a bulk of items at once. Moreover, it is possible to alter the date stored on the tag. This makes the automation of the systems easier and reduces time, man-power and costs (see Dong-Liang et al. [3]). Department stores for example could reduce their out-of-stock items by tagging them with RFID chips (see Roberti [2]). In general EPC is seen as the successor of the barcode, which not only holds information about the type of good, but also a serial number, which uniquely identifies the object (see Dong-Liang et al. [3]). EPC tags attached to items could also be used as an anti-theft measure (see Yuan and Huang [16]).

- **Toll Collection**
  Automated electronic toll collection, is one of the earliest applications based on RFID technology and is used all over the world. Active tags are attached to vehicles, which can pass through the toll gate without the need to slow down. This reduces traffic jams and costs, that come along with human operated toll stations (see Dong-Liang et al. [3]).

- **Animal Identification**
  LF tags are used for farm animals as well as for pets. In large collaborative ranches RFID tags are used to distinguish the ownership of animals. They are also utilized to avoid the spread of diseases like mad-cow disease. Therefore the use of RFID technology is advantageous, because it can help to speed up the process of identifying and tracking the path taken by farm animals (see Jeffries [5]). When treating animals with hormones and medicines, the danger of giving an animal an overdose can be reduced with the usage of RFID tags (see Violino [13]). On modern farms, RFID readers could be used for automated and individual feeding and surveillance of the animals, which can cut costs and increase productivity.

- **Access Control**
  Plastic cards with integrated tags can be used to grant somebody a specific access. In big city public transport, such cards are used as electronic tickets. Moreover, ski resorts issue ski cards for the entrance of a lift or a cable car. Another example are exhibitions, entertainment venues and stadiums where RFID tags in e-tickets can reduce the threat of fake tickets (see Weis [1]). At big events like the FIFA World Cup and the Olympic Games, RFID tickets were used in large amounts, to additionally give better protection against counterfeit tickets, as in the past. The same is true for other events like World Expos or concerts where such tickets are in use (see Dong-Liang et al. [3]).

- **Contactless Payment**
  Contactless Payment based on RFID technology can be achieved by different approaches like credit cards, smart cards, key fobs or mobile phones. Thereby allowing the electronic money to be either stored on the device itself, or making it possible

for the device to invoke a transaction from the customer's account to the seller's account. The most important application in this field is the contactless smartcards. The main international financial cooperations, like Visa, MasterCard and American Express have implemented contactless RFID systems that are based on the existing infrastructure for the current payment cards with a magnetic strip (see Lacmanovic et al. [4]).

RFID technologies enable a lot more applications, besides those mentioned above. Having in mind the continuing development of RFID technology, especially for EPC tags in the field of item tagging, one could imagine even more applications for RFID tags, which could increase efficiency or make everyday life more comfortable. By having items and goods with RFID tags on them, a household could be managed more efficiently. Devices like refrigerators, washing machines etc. could work more intelligently, depending on the information the items have to handle.

2.3 Security and Privacy Challenges of Tags

Having RFID tags in so many aspects of our life, does not only bring benefits. The possibilities that come along with the applications using RFID technology can go beyond their original intended assignment. Therefore the use of RFID technology can be a serious threat for a potential violation of our privacy and confidence in authenticity of the items or entities using RFID technology. In the context of RFID it is basically possible to distinguish between two main security issues: privacy and authenticity.

2.3.1 Privacy

RFID tags attached to personal items respond to RFID readers without being noticed by their owners. Thus an adversary could secretly inventory or track a tag attached to an owner. Even if the tag has no information about the individual, a random number or ID could be used to track a certain person (cf. Figure 2.1), once it is associated with that person (e.g., when buying an item cashless with a smartcard).

A product with an EPC tag may contain not only a serial number, but also information about the product itself. So an adversary could gain information about private and sensitive data of a person. For example if someone is wearing her or his medicine in a bag, one could gain information about that. With this information an adversary could guess which medical conditions the person suffers from. As long as there is a continuing growth in the use of RFID technology and as long as it is not that widespread, it is hard to imagine all the possible threats that come along (for examples on how RFID could be used in the future see Juels [6]).

It is also important to mention that in comparison with other technologies, like mobile phones, where tracking is also a serious concern, the required hardware utilities for tracking RFID tags are cheaper and easily accessible. For example, to track a mobile phone one would need access to the network of the operator. The easy access to RFID technologies makes the threat of these scenarios appear more likely and real.

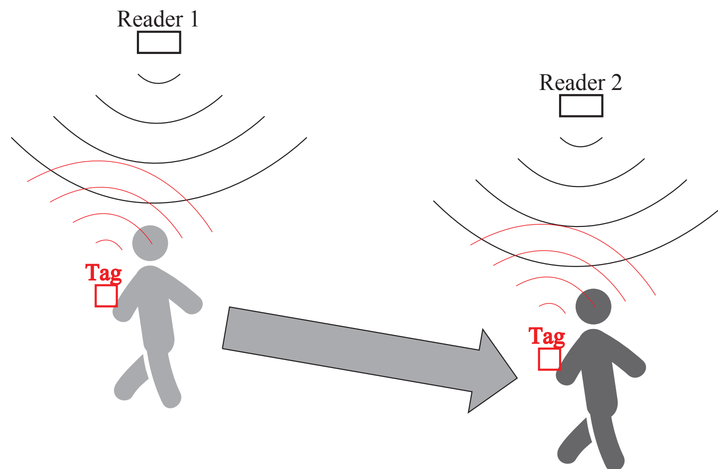

Figure 2.1: Person being tracked, by wearing a tag with a specific ID

But also in RFID applications, where privacy is a basic requirement of the application itself, many of the current systems have insufficient security functionality. For example, the E-passport introduced by the International Civil Aviation Organization's (ICAO), holds personal information about their owners. Over the years three generation of the E-passport were released. It was shown that the first two generations had serious security weaknesses (see Bogari et al. [8]). Bogari et al. [8] mentions that there could also be a security risk in the third generation of the E-passport.

Thus, there are already a number of organizations which express their concerns about involving our lives with a technology that doubtlessly could facilitate our lives, but also bare many risks (see CASPIAN [17]).

2.3.2 Authenticity

At first glance, an uninformed user might think an RFID tag that identifies a good is proof of origin for the good. In fact this is a bit too frivolous. A basic RFID tag without any security functionality is not trustworthy at all. The ID of a tag can easily be read by an adversary's reader and be programmed into a counterfeit tag. The first version of the EPC Gen2 standard that is expected to replace the bar code and already in widespread use, does not offer any instruments to counter such attacks. Minimalistic approaches towards counter measures, use digital signatures to prove the authenticity of the data on a tag. However, this is only a protection against fake data on a tag, it does not prevent the copying of the data to clone the tag. Such counterfeit tags could be used to prove the authenticity of a fake good that are cheap copies of the original ones. Other possible applications for counterfeit tags are to prove the identity of an individual and trigger a certain event, without the real individual being aware of that. For example, a clandestine reader could spy on the unlocking mechanism of a car, and allow a burglar to open the car with a fake tag (cf. Figure 2.2).

Other counter measures against counterfeit tags are based on the fact that a tag has a unique serial number or identifier. In a closed system where a centralized database has knowledge of the data of all tags at every point in time and place, it is possible to detect two identical tags. Then the operator of the database would have an indication of forgery. Nevertheless, systems with such conditions, were it is possible to observe all the existing tags of the system, are rare. Either it is not possible to observe all the tags at every

location, or the authentic tag and the fake tag are read at different times, what makes them indistinguishable.

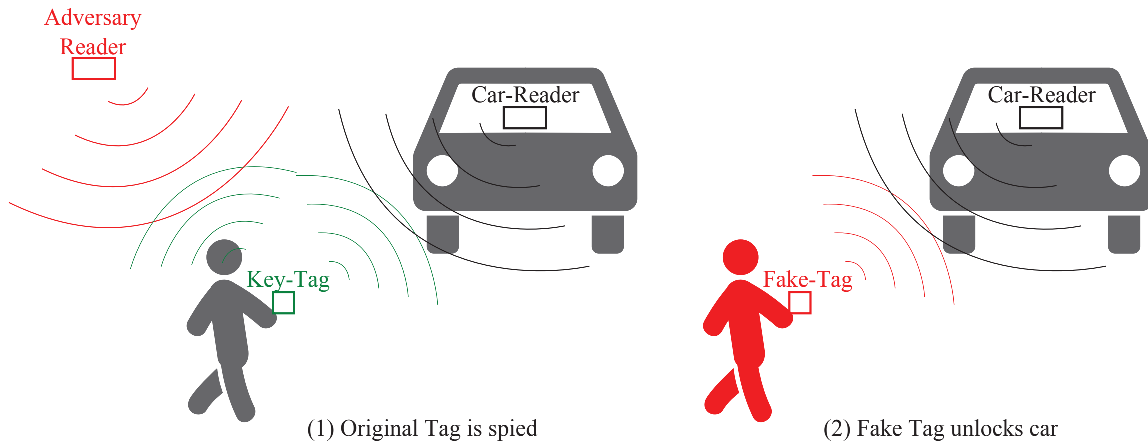

Figure 2.2: In (1) the original car-unlocking tag is being spied on. In (2) the spy broke the security features of the tag and cloned it

Authenticity is not only a concern in RFID applications. In history there have already been some investigations into security systems that prove the authenticity of data. Although there should be knowledge about the security risks of some systems, there are a bunch of solutions on the market that have much too weak protections in place (see Juels [6] and Pateriya and Sharma [7]). The Digital Signature Transponder (DST) for example is a device used to authenticate the owner of the device that includes the DST. Amongst other applications it is implemented in cars unlocking systems and also in tags for contactless payment. It is shown that the security functionalities in DST are not strong enough and vulnerable. The weakness in DST is based on the short key length of 40 bits. In Bono et al. [18] it is described, how the security of DST can be broken and how it is possible to start a car or buy gasoline by using a simulated DST device.

2.4 Security-Efforts on EPC-Tags

As privacy and authenticity issues are not something new to RFID systems, there have been quite a few investigations towards these issues by academic institutes as well as by the industry. Because costs are the limiting factor for RFID devices, by means of financial costs as well as costs in power consumption, much research has been conducted to try to find architectures that would possibly reduce the costs. In Juels [6] there are a few proposals summarized that do not implement traditional cryptography, but "workarounds" which extend the functionality of basic tags in a minimalistic way.

The simplest way to avoid clandestine tracking of an EPC tag is to kill the tag when it is not needed anymore for its original intended use, and thus disable it's reply to readers. This might be a solution for many scenarios, but would also preclude some benefits one could have with an EPC tag that remains active after its original intention. For example returns, recalls and recycling of goods could be optimized by having some of the original information of a tag available. An other approach is to *rename* the tag. In Juels [6] several methods for renaming, changing or (re-)encrypting[1] the EPC of a tag, but still leaving it

[1]Re-encryption stands for the repeated encryption of the ciphertext.

identifiable, are staged.

However, authenticity is harder to achieve. The EPC Gen2 standard[1] does not provide real mechanisms for tag authenticity, thus cloning a tag does not require much effort. Various proposals for RFID tags with security are related to the utilization of strong cryptographic functions like AES. In Man et al. [19] and Ricci et al. [20] it is shown that such tags can achieve an acceptable low-power consumption.

Nevertheless, achieving authenticity is not only a question of a strong cipher. Moreover, the proper protocol has to be implemented to ensure a secure communication. In 2010 the International Organization of Standardization (ISO) suggested the working draft ISO/IEC WD 29167-6 (see ISO/IEC WD 29167-6 [21]). Three security protocols are described there, which should strengthen the security of the ISO/IEC 18000-6 standard (see ISO/IEC 18000-6 [15]), and thus the security of EPC tags. One of the protocols, denoted as **Protocol 1**, in the working draft ISO/IEC WD 29167-6 (see ISO/IEC WD 29167-6 [21]) is based on an AES-128 cipher, which is generally accepted as a strong cipher. The goal of **Protocol 1** is to allow mutual authentication between an tag and a reader. Not much later after it was released in Bagheri et al. [9] and Song et al. [10] it was shown that **Protocol 1** is vulnerable to man-in-the-middle (MITM) attacks. In a MITM attack an adversary eavesdrops on the communication between the parties and manipulates the messages they send each other. Thereby the parties do not recognize the manipulation and accept the corrupted message. In the proposed attack against the **Protocol 1** an adversary is able to manipulate the messages in such a way that a tag and a reader would successfully authenticate each other, but every subsequent command from the reader to the tag would fail.

This example shows that although an adequate cryptographic primitive is used[2], the security functionality still can fail.

[1] The 2nd version of the EPC Gen2 standard, which was released during the work on this thesis, has additional commands that provide a framework for certain security goals.

[2] 'Adequate' in comparison to the weak primitive in DST that has a short key length, mentioned in section 2.3.2.

Chapter 3

Security in Communication

Historically one of the first intentions for security was to hide information during a communication process. Centuries ago, the first security approaches had less scientific background, but tried to distort the information in a way that an adversary could not understand it. In the beginning of the last century, especially during the world wars, different nations and their militaries massively investigated new cryptographic techniques. On the one hand they wanted to invent systems that would protect their communication process, on the other hand they tried to break the security of their 'enemies'. Since those investigations were done by national agencies, the public had limited knowledge and access to them.

With the development of digital communication in various applications of people's life (e.g., phones, e-mail, fax, etc.) security has become the main concern for a lot of people. Not only communication but also other aspects of our life have become affected by the development of digital technologies, like E-money, e-voting, e-government etc. Thus in the end of the last century there was a rising research in security and cryptography.

For more details about security and security applications see Menezes et al. [22], Anderson [23] and Schneier [24].

3.1 Cryptographic tools

In Menezes et al. [22] cryptography is described as following:

> "Cryptography provides techniques for keeping information secret, for determining that information has not been tampered with, and for determining who authored pieces of information"

. Further in Menezes et al. [22] and other literature like Schneier [24] the basic objectives of cryptography are mentioned as[1]:

- **Confidentiality**. The information should be only accessible by individuals who are meant to do so.
- **Authenticity**. An individual must be able to verify the origin of a piece of information. No adversaries should be able to fake the origin or reuse the information as their own.

[1] Other goals can be derived from the listed objectives

- **Integrity**. An individual must be able to ascertain that the information has not been manipulated. No adversary should be able to modify any information by means of inserting, deleting or substituting certain parts of the data.
- **Non repudiation**. Information once committed by an individual should not be deniable by the latter afterwards.

The cryptographic primitives for achieving those goals can be basically distinguished in three different groups:

- Symmetric Crypto Primitives
- Asymmetric Crypto Primitives
- Hash Functions or Unkeyed Primitives

For the first two groups the concept of an encryption scheme (or *cipher*) is essential. Although for the purpose of a hash function it is not required, also many hash function are based on a cipher. According to Menezes et al. [22], an encryption scheme is a set of encryption transformations E_e and corresponding decryption transformations D_d, depending on the key pair (e, d), with the property $D_d = E_e^{-1}$. Further an encryption transformation E_e is a certain bijection from the plaintext (or message) m to the ciphertext c, depending on the key e. A decryption transformation is a bijection from the ciphertext c to the plaintext m, depending on the key d. The keys e and d are elements of a defined set, the so called *key space*. A *cipher* complies $D_d(E_e(m)) = m$ for all possible messages. Figure 3.1 gives an overview over an encryption scheme.

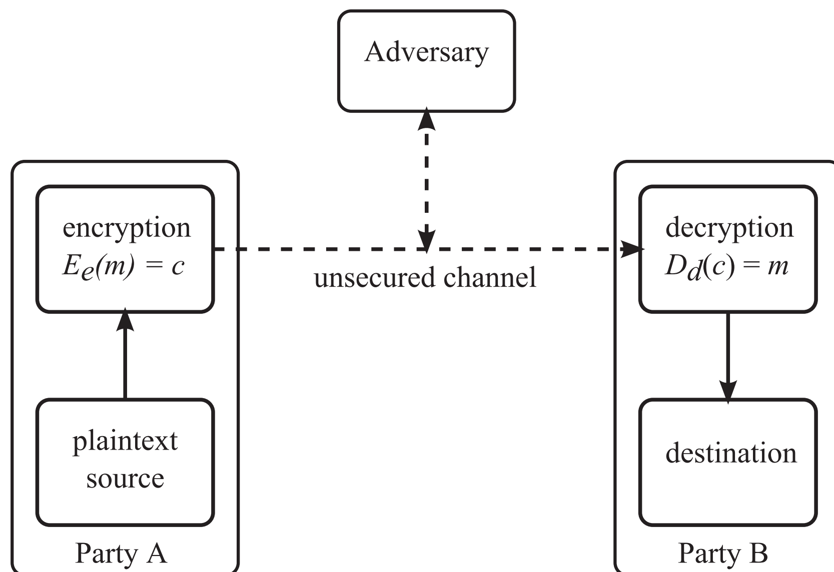

Figure 3.1: Schematic of an encryption scheme used by two parties and an adversary. Adapted from Menezes et al. [22].

Independent of the primitive, for a secure scheme it is important that the transformations E_e, D_d and the *key space* can be known in public. Security should be based on the secrecy of the used key pair (e, d). The approach *security through obscurity*, where the security is based on the secrecy of the implementation of the transformations, is very controversial in cryptology (e.g. Yu and Brune [25]).

3.1.1 Symmetric Crypto Primitives

In symmetric encryption schemes, the key d can be easily derived from the key e and vice versa (in most schemes $e{=}d$). Before the communication between two parties start, they have to agree on a key over a secured channel. An adversary should not be able to reorder, delete, insert or read the information from a secured channel.

Symmetric encryption schemes that operate bit wise on the plaintext are called *stream ciphers*. Encryption schemes that operate on parts of the plaintext, so called blocks, are called *block ciphers*.

3.1.1.1 Block Ciphers

Block ciphers operate on a fixed amount of data. They map parts of the plaintext, with the block length n to a ciphertext of the same length. This mapping could be seen as a simple substitution function, where whole character blocks are substituted by blocks of the same size. This substitution function should be variable, depending on a secret key with the key length k. This characteristic would also apply to block ciphers in asymmetric cryptography.

Symmetric block ciphers divide the data into smaller parts, which are then subsequently combined and transformed. These transformations are mostly based on substitution and permutation and build so called *substitution-permutation (SP)* networks. Since the phase of substitution and permutation builds an internal function that is repeated numerous times, ciphers based on this principle are also defined as an *iterated block cipher*. Figure 3.2 shows the principle of an iterated substitution-permutation network.

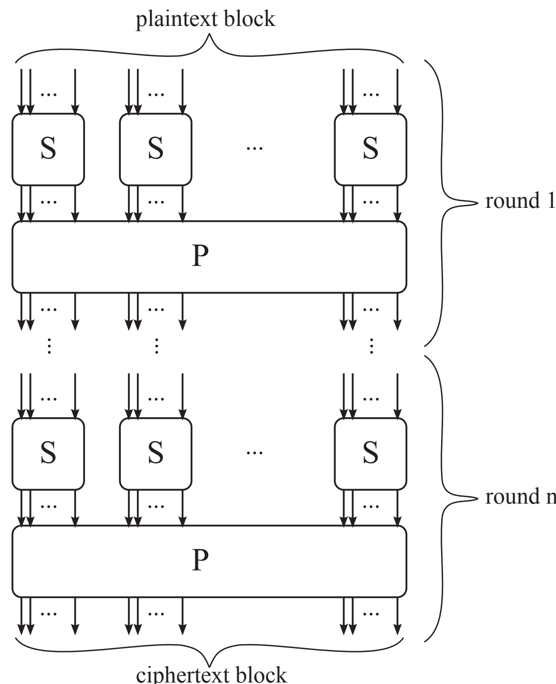

Figure 3.2: An iterated substitution-permutation (SP) network.

Since the plaintext often is much larger than the block size of an block cipher, it has to be split into blocks of the according size. When enciphering those blocks, identical

plaintext blocks would lead to identical ciphertext blocks. This could be a weakness to attacks that are based on the frequency distribution of characters and strings in the plaintext. Thus a proper *mode of operation* should be used that links the particular ciphertext blocks.

Applying the cipher block by block, as depicted in Figure 3.3, is called an Electronic Code Book (ECB) mode. Here, no kind of chaining or feedback of the particular blocks is used and thus identical plaintext blocks lead to identical ciphertext blocks. In the Cipher Block Chaining (CBC) mode, as depicted in Figure 3.4, a certain ciphertext block is linked with the succeeding plaintext block. For the first encryption step an initial value is needed. In Output-Feedback (OFB) mode , as depicted in Figure 3.5 an initial value is encrypted in the first step. Then the output of the current step is taken as input for the next step. The plaintext block is only linked with the output of the block cipher. Thus, in comparison to the CBC mode, the stream that is applied on the plaintext, can be precalculated.

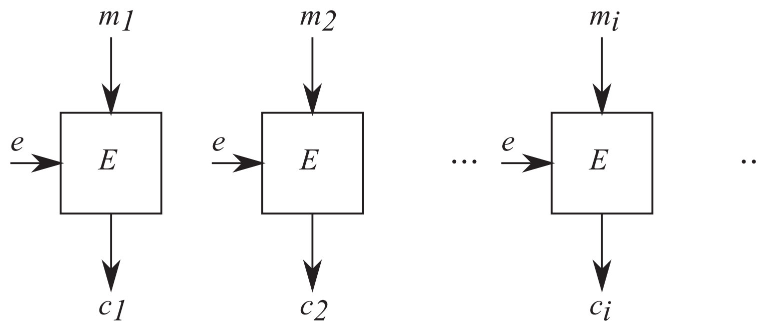

Figure 3.3: Electronic Code Book (ECB) mode

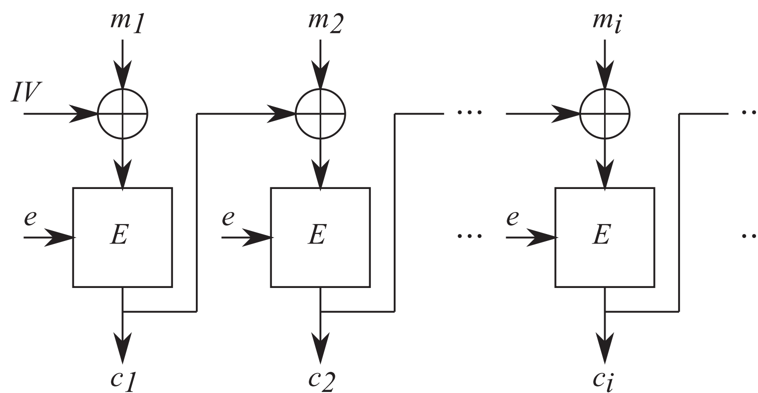

Figure 3.4: Cipher Block Chaining (CBC) mode

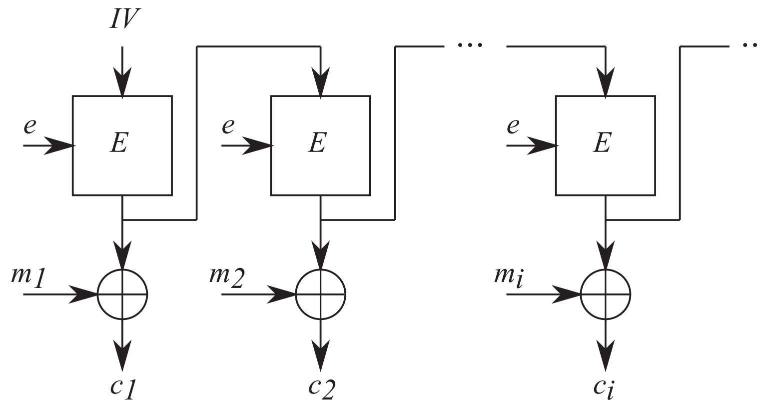

Figure 3.5: Output Feedback (OFB) mode

In the 1970s the Data Encryption Standard (DES) algorithm was the first commercial symmetric cipher to be widely used. But early on, there were serious concerns about the security of this algorithm, mainly because of its short key length of only 56 bits. Also the design criteria of DES, which was designed by IBM, were kept secret for a long time. In the 1990s, when differential cryptanalysis was invented, the choice for the specific DES characteristics could be approved. The original designers admitted that they knew about the differential cryptanalysis techniques back in the 1970's. Nevertheless, already in the 90s so called cracking machines were developed which can apply an exhaustive key search in a few days (see Schneier [24]). Thus the National Institute of Standards and Technology (NIST) announced a contest, where it choose an algorithm out of several proposals for the new Advanced Encryption Standard (AES), which is discussed in Section 3.2.

3.1.1.2 Stream Ciphers

A stream cipher usually operates bit-wise on the plaintext. The *Vernam Cipher* is a very simple and effective implementation of a stream cipher. For every bit of the plaintext an *exclusive-or* operation (XOR) is applied with a bit from a key stream sequence. If this key stream is truly random and never used again, than the Vernam Cipher is called a *one-time-pad*. For practical reasons a random stream is not feasible, and thus the stream is produced by an algorithm that is called the *keystream generator*.

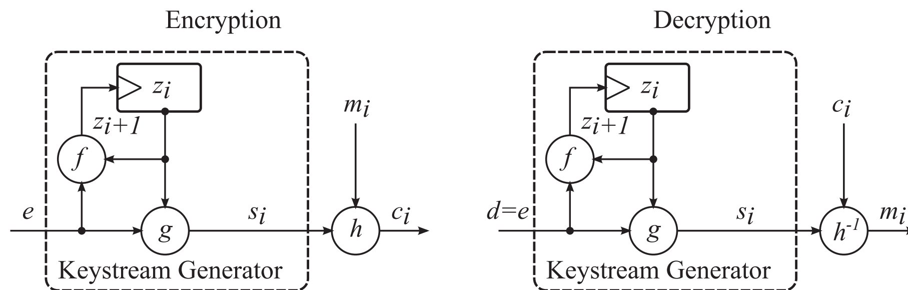

Figure 3.6: General scheme of a synchronous stream cipher with the key e, the ciphertext c_i, the plaintext m_i, the keystream s_i and the internal state z_i

If the key stream is generated independently of the plaintext and of the ciphertext, than it is called a *synchronous* stream cipher. According to the general scheme of a

synchronous stream cipher in Figure 3.6, in the *Vernam Cipher* the output function h would be an XOR operation.

A very popular mechanism that is used for key stream generators are feedback shift registers (FSR). A special case are the linear feedback shift registers (LFSR). A Feedback Shift Register (FSR) consists, as the name says, of a shift register and feedback function. The feedback function states how the next bit that is shifted into the register bank depends on the current values of the registers. If the feedback function is a linear polynomial, then it is called a Linear-Feedback-Shift-Register (LFSR). This polynomial is then called the *connection polynomial*.

Figure 3.7 illustrates a general FSR. According to Menezes et al. [22] Figure 3.8 illustrates a LFSR with the connection polynomial $C(D) = 1 + c_1D + c_2D^2 + \ldots + c_ND^N$. Assuming that the initial state of the registers b_0, b_1, b_2, ..., b_{N-1} in Figure 3.8 is (s_0, s_1, s_2, ..., s_{N-1}) the resulting output stream for the LFSR is $s = s_0, s_1, s_2, s_3, \ldots$ and would be determined by $s_k = c_1s_{k-1} + c_2s_{k-2} + c_3s_{k-3} + \ldots + c_Ns_{k-N}$ for $k \geq N$. The secret key in a LFSR would be represented by the initial state of the registers or by the initial state of the registers and the connection polynomial, more precisely by its coefficients $(c_1, c_2, \ldots c_N)$, together (see Menezes et al. [22]).

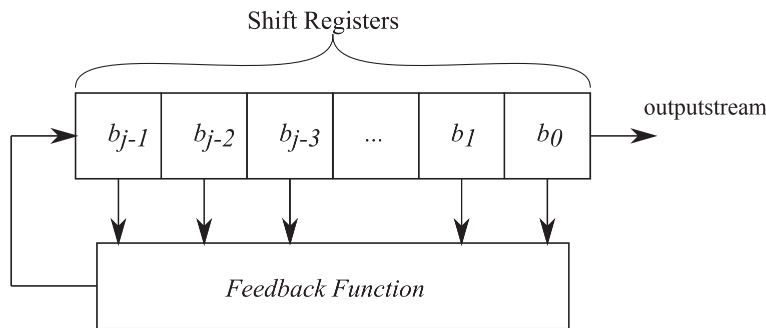

Figure 3.7: Scheme of a general FSR

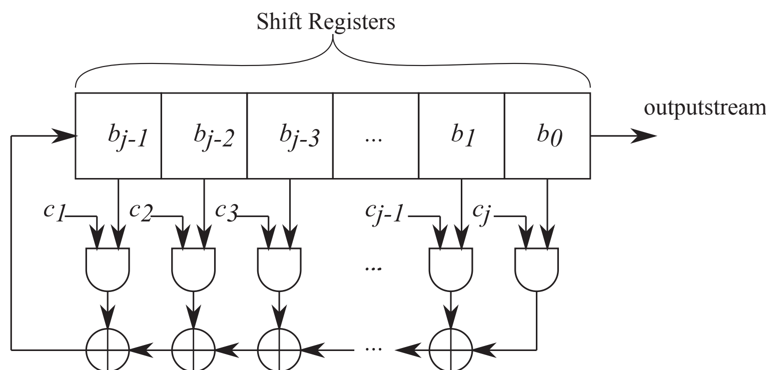

Figure 3.8: Scheme of a general LFSR with the connection polynomial $C(D) = 1 + c_1D + c_2D^2 + \ldots + c_ND^N$

3.1.2 Asymmetric Crypto Primitives

In asymmetric encryption schemes the encryption transformation has the property that knowing E_e, the encryption key e and the ciphertext c it is infeasible to find the message m

such that $E_e(m)=c$. Further, this means that it is infeasible to determine the decryption key d using the encryption key e. Before the communication between two parties starts, the receiving party has to transmit the encryption key e over any (eventually unsecured) channel to the transmitting party. The decryption key d remains a secret that is only known by the receiving party. Thus asymmetric encryption schemes are also known as *public-key* schemes. The encryption key e and the decryption key d are then called *public key* and *private key*.

For the purpose of an asymmetric encryption scheme a *trapdoor one-way function* is needed. A function $y = f(x)$ is a *one-way function* when it is easy to calculate the function $y = f(x)$, but it is computationally infeasible for essentially all y to find any x such that $f(x) = y$. The *trapdoor* is a certain piece of information that makes it feasible to find any x, given any y. In an asymmetric encryption scheme the *one-way function* $y = f(x)$ would correspond to the encryption transformation $c = E_e(m)$ with the key e. The *trapdoor* would be the decryption key d that is needed to compute the decryption $m = D_d(c)$. Many of these trapdoor one-way functions, and thus the asymmetric encryption schemes, are based on computational problems, such as the *integer factorization problem*. The *integer factorization problem* says that for any positive integer n, finding its prime factors $(p_1, p_2, ..., p_n)$, such that $n = p_1^{e_1} \times p_2^{e_2} \times ... \times p_n^{e_n}$ with $e_i \geq 1$ is computationally hard.

One of the most widely used encryption scheme in *public-key* encryption is the RSA cryptosystem. RSA is named after its inventors R. Rivest, A. Shamir and L. Adleman who proposed this encryption scheme in Rivest et al. [26]. In RSA the key pair e, d for encryption and decryption is derived as follows.

1. Generate two large prime numbers p and q, with almost the same amount of digits.
2. Calculate the products $n = p \times q$ and $\Phi(n) = (p-1) \times (q-1)$.
3. Choose an integer e (*public exponent*), with $1 < e < \Phi$ and $gcd(e, \Phi) = 1$.
4. Calculate the (unique) integer d, with $1 < d < \Phi$, such that $e \times d \equiv 1 (\bmod \Phi)$, using the extended Euclidean algorithm (see C.1).
5. The public key is (n, e). The private key is (n, d). Also the parameters p, q and $\Phi(n)$ should be kept secret, since they can be used to calculate d

The integer e is also called the *encryption exponent*, the integer d is called the *decryption exponent*. Given the public key (n, e) and the plaintext m the ciphertext c can be calculated as:

$$c \equiv E_e(m) \equiv m^e \bmod n \tag{3.1}$$

The plaintext m can be recovered, using the private key (n, d), as:

$$p \equiv D_d(c) \equiv c^d \bmod n \tag{3.2}$$

3.1.3 Hash Functions

A hash function *h(x)* is a function that maps data x with arbitrary length to data y with fixed (usually smaller) length. The smaller data with fixed length is called the *hash value* (or *image*). If the hash value is n-bit long, then the probability of computing a certain hash value y for a given message x shall be 2^{-n}. There are three properties stated in Menezes et al. [22] which should be fulfilled by a proper hash function.

- *pre-image resistance* - It is computationally infeasible, given a certain hash value y, to find an input x, the so called *pre-image*, such that $y=h(x)$
- *2nd pre-image resistance* It is computationally infeasible, given any input x_1, to find any second input x_2, the so called *2nd pre-image*, which has the same output as x_1, i.e. to find $x_2 \neq x_1$ such that $h(x_2) = h(x_1)$.
- *collision resistance* - It is computationally infeasible to find two inputs (x_1,x_2), such that $h(x_1)=h(x_2)=y$. Finding such two inputs (x_1,x_2) would be called a *collision.* (In comparison to the *2nd pre-image resistance* here both inputs are freely selectable)

The usual idea of a hash function is to give a representative image, a so called *fingerprint*, of the input data. Assuming that a hash function has the mentioned properties, the *fingerprint* can be used instead of the original data, as it is uniquely identifiable with the input data. The most common use of hash functions is as part of digital signatures or for data integrity. In most digital signatures a long message is hashed in the first step and afterwards the hash value is signed. The purpose for this is that signing the original message would require a higher computational effort for many of the digital signature schemes. If a party wants to verify the message, it also has to hash the original message in the first step. Then the party uses the verification transformation, which takes the hash value and the signature as input, to verify the message. This saves time and computational costs, since a message could be of arbitrary length. For security reasons it is important that it is infeasible to find a collision. Otherwise a party could calculate two different messages with the same hash value, sign the message x_1, and later claim that it signed message x_2 since the hash value would be the same.

For data integrity it is important to assure that a certain piece of data was not manipulated. Therefore the data that should be protected is hashed at a certain point in time. The integrity of this hash value has to be protected with a proper technique, e.g. through an authentic communication channel. If a party wants to check if the data was manipulated, it first calculates the hash value of the data that is transmitted through a potentially unsecured channel and then compares it to the original hash. Hash functions used for data integrity and digital signatures typically do not have any kind of keys, thus called unkeyed hash functions.

3.2 Advanced Encryption Standard

In 2001 the NIST chose the Rijndael cipher (see Daemen and Rijmen [27]) for the new Advanced Encryption Standard AES. For the design of the AES cipher, security as well as efficiency were equal criteria. Though the original proposal specified also other key and block sizes, AES works on a block size of 128 bit with keys lengths of 128, 192 or 256 bits. Depending on the key length, the AES cipher has either 10, 12 or 14 rounds of computation.

AES works on an *internal state S*, which equates to a $4{\times}4$ byte array and starts with the plaintext as the initial state. After an initial round every other round has four different transformations. Each of it has its own function. These transformations could be assigned to one of the following three layers, which provide a certain resistance against linear and differential attacks (see Daemen and Rijmen [27]):

- The **linear mixing layer** should maximize the diffusion over the rounds. This applies to the *ShiftRows* and *MixColumns* transformations of AES.

- The **non-linear layer** should maximize the nonlinear degree. This applies to the *SubBytes* transformation of AES.
- The **key-addition layer** makes every round dependable on the secret key. This applies to the *AddRoundKey* transformation of AES.

The *SubBytes* transformation in Figure 3.9 replaces every byte from the byte array (the internal state) with a byte from a lookup table. The lookup table corresponds to a multiplicative inverse over the galois field $\mathbf{GF(2^8)}$ combined with an affine transformation.

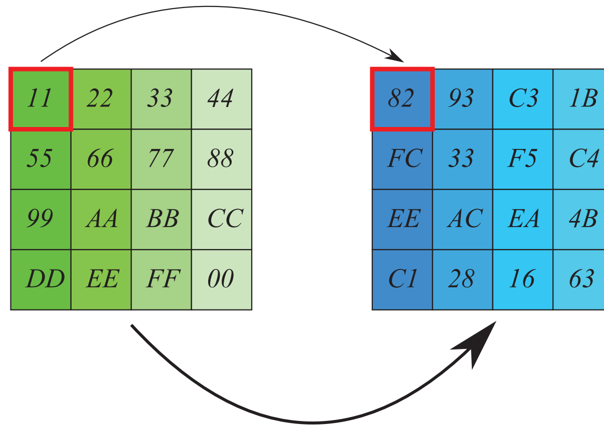

Figure 3.9: Example of the *SubByte* transformation on the *internal state* S.

Figure 3.10 shows how the *ShiftRows* transformation operates on the internal state S. This transformation leads to a high diffusion over the columns.

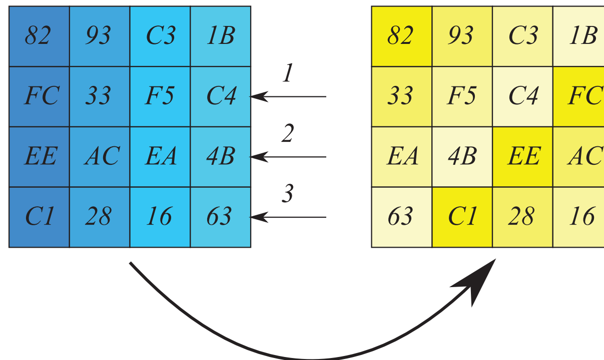

Figure 3.10: Example of the *ShiftRows* transformation on the *internal state* S.

The *MixColumns* transformation in Figure 3.11 is based on a matrix transformation over the galois field $\mathbf{GF(2^8)}$. It leads to a high diffusion over the state through a single column.

Algorithm 1 AES round transformation

1: **procedure** ROUND($State, RoundKey$)
2: SubByte($State$)
3: ShiftRows($State$)
4: MixColumns($State$)
5: AddRoundKey($State, RoundKey$)
6: **end procedure**

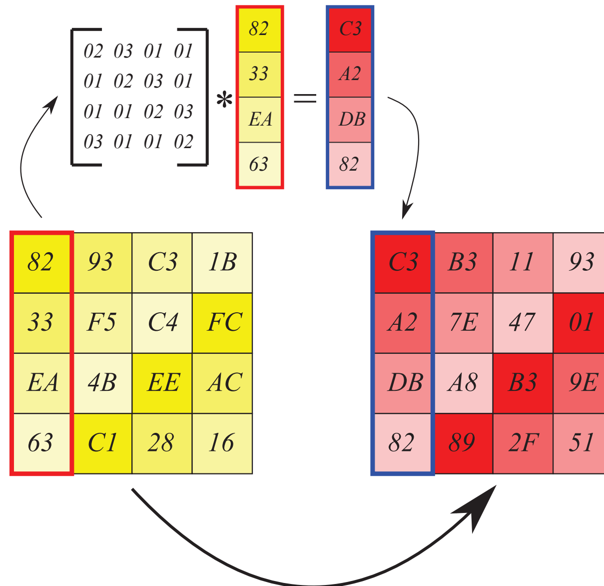

Figure 3.11: Example of the MixColumns transformation on the *internal state S*.

The *AddRoundKey* transformation, shown in Figure 3.12, applies an XOR operations on the internal state with the round key. Every round key is computed out of the previous round key (the first round key is computed out of the secret key) by an particular key schedule. This key schedule, as described in Daemen and Rijmen [27, sec. 4.3], can be done independently and prior to the other transformations. Given these transformations, an AES round can be defined as stated in Algorithm 1.

Algorithm 2 AES cipher

1: **procedure** AES($State$,$ExpandedKey[RoundCount + 1]$)
2: AddRoundKey($State$,$ExpandedKey[0]$)
3: **for** $i = 1$ to $RoundCount$ **do**
4: Round($State$,$ExpandedKey[i]$)
5: **end for**
6: SubByte($State$)
7: ShiftRows($State$)
8: AddRoundKey($State$,$ExpandedKey[RoundCount + 1]$)
9: **end procedure**

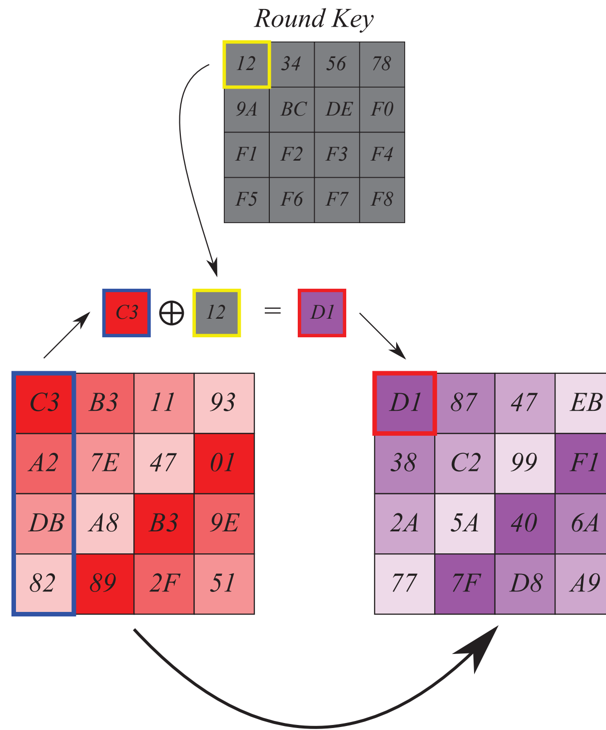

Figure 3.12: Example of the AddRoundKey transformation on the block state

Depending on the key size, AES has 10, 11 or 12 round transformations. The last round transformation is different in comparison to the others, as the *MixColumns* transformations is skipped. Also before the first round transformation, an *AddRoundKey* transformation is applied, because starting the cipher with the *SubByte*, *ShiftRows* and *MixColumns* transformation without applying any secret key is cryptographically worthless and would only produce computational overhead. Assuming that the round keys are precalculated from the original cipher key the whole cipher is stated in Algorithm 2. The array *ExpandedKey* is an array of 16-byte words, representing a block of the round key as shown in Figure 3.12 and containing the particular round keys. For more details about the round transformations and the key schedule see Daemen and Rijmen [27].

3.3 Protocols

The proper use of cryptographic primitives, which assure certain goals for different applications, is specified in protocols. In Schneier [24] a protocol is called

> "...a series of steps, involving two or more parties, designed to accomplish a task"

This means that a protocol has a sequence with a defined beginning and an end. The steps involve either a computation or a transmission or a reception of a certain message. A protocol requires at least two communication parties, since communication without a partner does not make sense. The parties have to follow some rules in a defined order to perform the protocol, and thus finish it as appointed. Finally, the protocol has to achieve something, like a proof or an exchange of information, otherwise it is useless.

Other important characteristics for a protocol are also that (i) everyone involved has to know all the steps of the protocol, (ii) everyone has to agree on the protocol, (iii) the protocol has to be unambiguous and (iv) the protocol must have a defined action for every possible situation.

In Schneier [24], depending on the function of the involved parties, it is distinguished between three types of protocols:

Arbitrated protocols. In an arbitrated protocol, as shown in Figure 3.13, a disinterested, trusted third party, the so called arbitrator, is required. Disinterested means that the arbitrator has no allegiance to any party involved in the protocol. Trusted means that all parties involved in the protocol trust that the arbitrator is acting correctly and honestly. Arbitrators in the real world are lawyers, public notaries and banks, and help parties that do not trust each other. Using an arbitrated protocol has always an extra delay and the arbitrator could become a bottleneck, since the third party has to deal with every transaction. Further, the arbitrator is required to complete the protocol, although he does not participate in every step of the protocol. Also, if the arbitrator is subverted, no one can trust him anymore. Having an arbitrator produces additional costs.

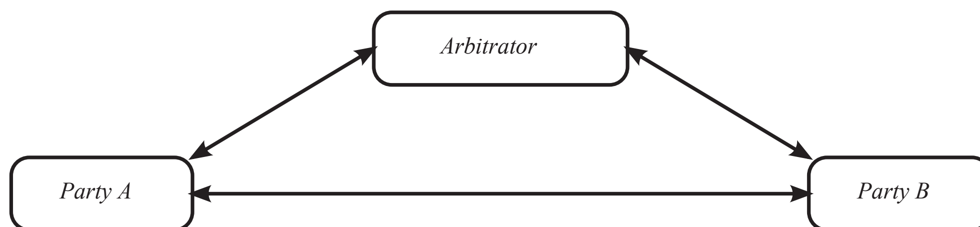

Figure 3.13: Three basic types of protocols.

Adjudicated protocols. In an adjudicated protocol, as shown in Figure 3.14, basically an arbitrated protocol is split into two parts. The non arbitrated part is executed when the parties are completing the protocol. The arbitrated part is executed only when there is a dispute. The arbitrator then is called an adjudicator, which in the real world would be represented by a judge. Two parties can enter a contract without a judge, but only call him when someone suspects cheating. So in a first instance an arbitrated protocol relies on the parties being honest, but in a dispute

the protocol provides an evidence for cheating and, depending on the protocol, also an identification of the cheater. Adjudicated protocols do not prevent cheating, but they can detect it. This shall discourage cheating, since in a proper protocol the detection is inevitable.

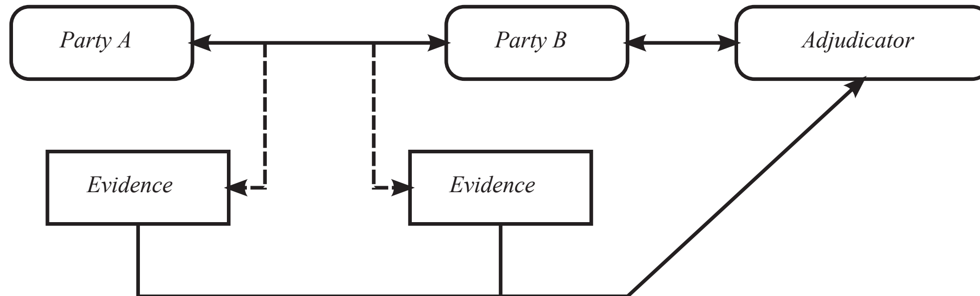

Figure 3.14: Three basic types of protocols.

Self-enforcing protocols. In a self-enforcing protocol, as shown in Figure 3.15, there is no need of an arbitrator or adjudicator. The protocol is designed in such a way, that no third party is needed to complete the protocol or to resolve a dispute. If one of the parties is somehow cheating, the other party detects this immediately and thus aborts the protocol. Self-enforcing protocols would be the preferred type of protocols but they cannot be established for every possible situation where security is needed

Figure 3.15: Three basic types of protocols.

Chapter 4

The EPC Gen2 standard

The electronic product code (EPC) is an international system used for identifying physical objects. Not only products, as the name suggests, but also animals, documents, industrial plants and even locations can be uniquely identified.

The EPC Gen2 standard EPCglobal Inc. [11] is a UHF standard for communication with a carrier frequency between 860 MHz and 960 MHz. It complies with the ISO 18000-C standard that defines the air interface for RFID devices in the mentioned frequency spectrum. The standard describes and defines the physical layer, the protocol and the commands for communication between an interrogator and a tag. Section 4.1 gives an overview how data is communicated between an interrogator and a tag. In Section 4.2 the mechanisms for a read and write access to the tags data are described. The following Sections are based on EPCglobal Inc. [11] and represent an excerpt that is relevant for the further work.

4.1 Signaling

In the physical layer, frequencies, modulation, coding and other communication parameters are defined. The communication between the interrogator and tag operates in half-duplex, which means that the tag does not simultaneously listen for commands from the reader and backscatters responses. Further, the system works as an Interrogator-Talks-First (ITF) system. In such a system, a tag responds only if it has previously received a valid command from the interrogator.

4.1.1 Communication from Reader to Tag

The reader uses a Double-Sideband Amplitude Shift Keying (DSB-ASK), a Single-Sideband Amplitude Shift Keying (SSB-ASK) or a Phase-Reversal Amplitude Shift Keying (PR-ASK) data modulation technique for sending signals to the tag. The signals are encoded in the Pulse Interval Encoding (PIE) format. Figure 4.1 shows the envelope for the data-0 and data-1 encoding for the modulation of the carrier frequency.

The distinctive characteristic of this encoding format is that a data-0 and a data-1 symbol do not have the same length. In this standard the *Tari* value is the reference time interval for a data-0 length. A data-1 symbol has 1.5 - 2 times the length of a data-0 symbol.

Each command is preceded by a *preamble* or a *frame-sync* before the first data bit.

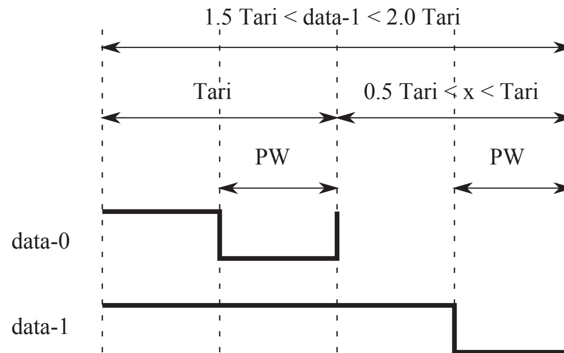

Figure 4.1: PIE symbols used to encode data-0 and data-1 for the interrogator to tag communication. Figure adapted from EPCglobal Inc. [11]

The *frame-sync* holds the **RTcal** value, which is a calibration value for the reader to tag communication and specifics the length of a data-1 symbol compared to the data-0 symbol. It represents the sum of both symbol lengths.

The *preamble* holds, in addition to the *preamble*, also the **TRcal** value, which is a calibration value for the tag to reader communication.

A *preamble* is preceded only before a *Query* command. For all other commands a *frame-sync* is preceded. The envelopes for these are shown in Figure 4.2. For further explanation of the values see EPCglobal Inc. [11].

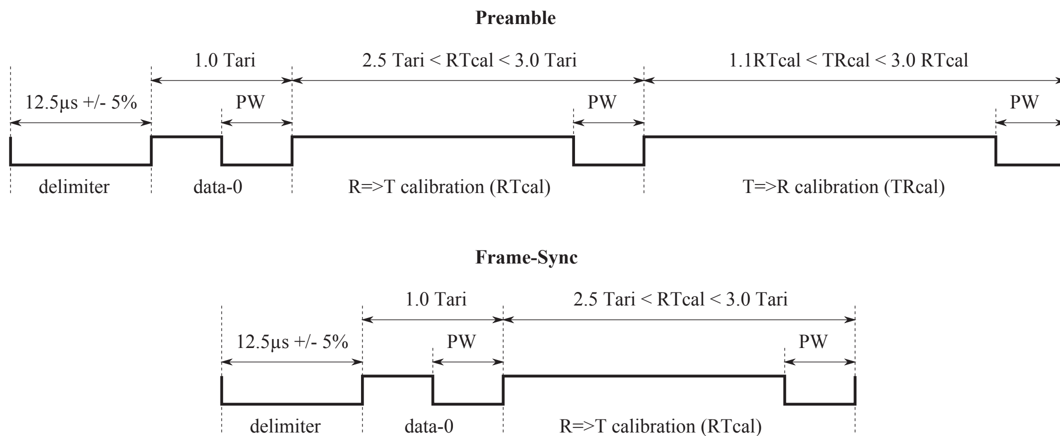

Figure 4.2: Preamble and frame-sync for the interrogator to tag communication. Figure adapted from EPCglobal Inc. [11]

4.1.2 Communication from Tag to Reader

The tag responds to the reader, by receiving an unmodulated signal from the latter and thereby changing its reflection coefficient at a certain clock rate. This produces sidebands that differ from the original carrier. The clock rate, the tag uses for changing its reflection coefficient, is called the Backscatter-Link frequency (BLF). It is calculated out of the **TRcal** value and the *DR* parameter of the *Query* command.

$$BLF = \frac{1}{T_{pri}} = \frac{DR}{\mathbf{TRcal}} \tag{4.1}$$

The data is modulated either in FM0 or Miller format. The format is selected by the interrogator through the M parameter of the *Query* command. Figure 4.3 shows sequences for the FM0 format. In this format, an edge between each symbol is present.

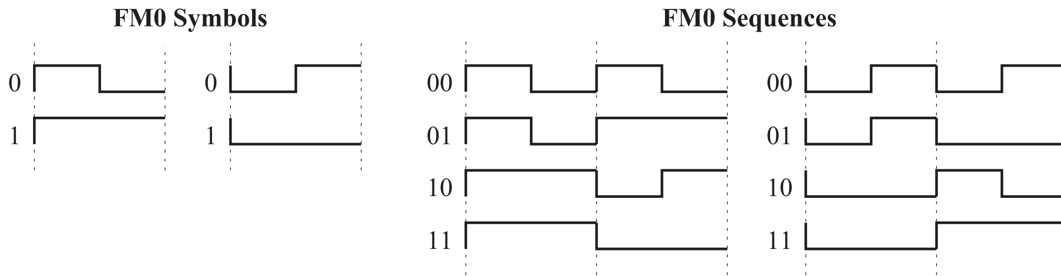

Figure 4.3: Examples on how to encode data-0 and data-1 in FM0 format for the tag to interrogator communication. Figure adapted from EPCglobal Inc. [11]

In the Miller format the phase is inverted either in the middle of a data-1 symbol, or between two data-0 symbols. Figure 4.4 shows the sequences for the Miller format at different cycles per bit.

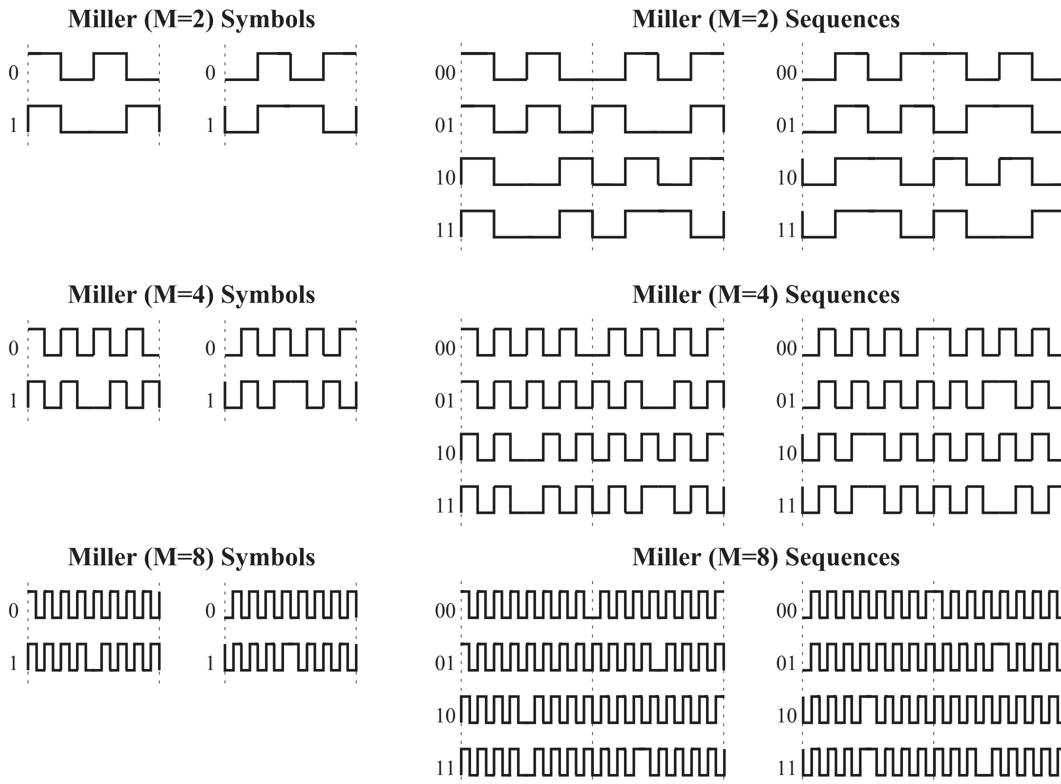

Figure 4.4: Examples on how to encode data-0 and data-1 in Miller format for the tag to interrogator communication at different cycles per bit. Figure adapted from EPCglobal Inc. [11]

In both, the FM0 and the Miller format, each data sequence ends with an unused data-1 symbol, which in EPCglobal Inc. [11] is stated as “dummy 1”. Each answer begins with a preamble sequence. The preamble depends on the current setting of the *TRext* flag, which is set for every inventory round by the initiating *Query* command. Figure 4.5 and Figure 4.6 show the preamble for FM0 and Miller encoding.

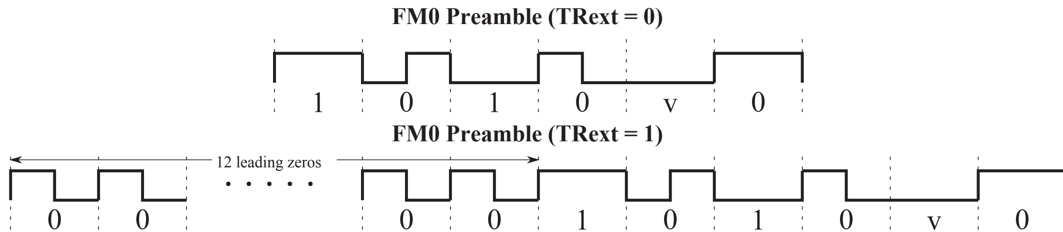

Figure 4.5: Preamble for the tag to interrogator communication in FM0 format. Figure adapted from EPCglobal Inc. [11]

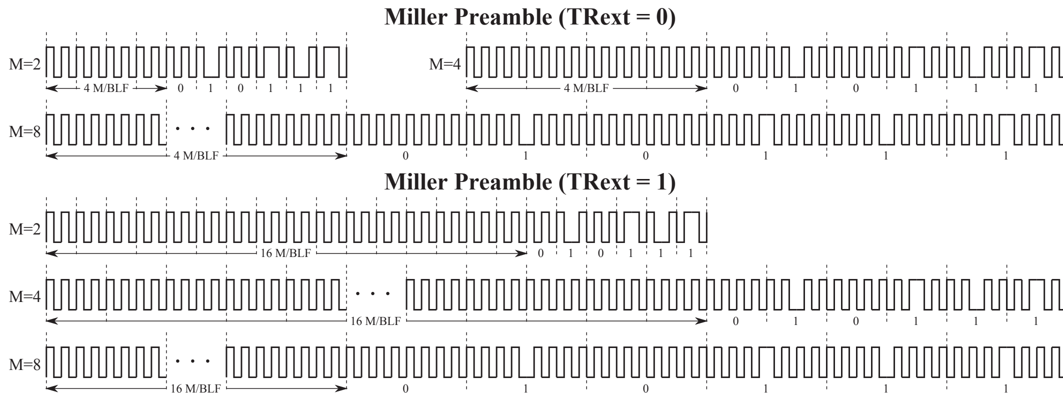

Figure 4.6: Preamble for the tag to interrogator communication in Miller format at different cycles per bit. Figure adapted from EPCglobal Inc. [11]

4.1.3 Link Timing

Table 4.1 lists the timing requirements for the interrogator to tag communication, which is illustrated in Figure 4.7.

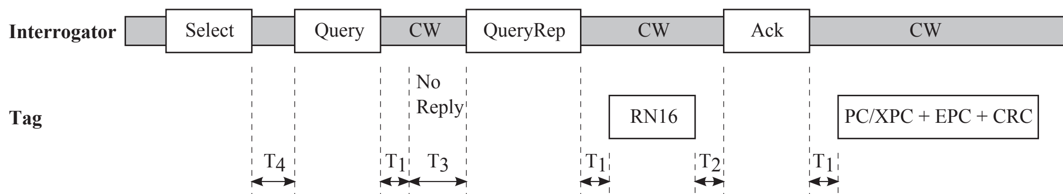

Figure 4.7: Link timing. Figure adapted from EPCglobal Inc. [11]

Especially the time T_1, the time from the ending of an interrogator transmission to the beginning of a tag response or receiving a new command, is a very critical value in this protocol.

The time T_1 is quite short in comparison to the logical decisions, that have to be calculated for a specific command. In the worst case it is 2.5 times the lowest possible Tari value, which is 2.5×6.25=15.63µs.

Moreover the effects on the tag and its response, depend on a valid command. Since the data length is important for a valid command, the required processing can not be made until another incoming symbol can be excluded. In worst case after the last valid symbol of a command, an invalid data-1 symbol could follow.

So, as illustrated in Figure 4.8, the minimum time for receiving a falling edge from a data-1 symbol has to be waited for until a command can be assumed finished with a valid amount of symbols. This delay of recognizing the end of a command is important in Section 5.6.3.

Since a memory write operation on a tag would take more than the specified T_1 time, a command that causes a memory write operation is allowed to exceed T_1.

Table 4.1: Link timing parameters. Table adapted from EPCglobal Inc. [11]

Parameter			Description
T_1	**Minimum**	max(RTcal,10T_{pri} ×(1- $\|FT\|$) - 2µs)	Time from the interrogator transmission to the tag response (specifically, the time from the last rising edge of the last bit of the interrogator transmission to the first rising edge of the tag response)
	Nominal	max(RTcal,10T_{pri})	
	Maximum	max(RTcal,10T_{pri} ×(1+$\|FT\|$) + 2µs)	
T_2	**Minimum**	3.0T_{pri}	Interrogator response time required if a tag is to demodulate the interrogator signal, measured from the end of the last (dummy) bit of the tag response to the first falling edge of the interrogator transmission
	Nominal		
	Minimum	20.0T_{pri}	
T_3	**Minimum**	0.0T_{pri}	Time an interrogator waits, after T_1, before it issues another command
	Nominal		
	Minimum		
T_4	**Minimum**	2.0 RTcal	Minimum time between interrogator commands
	Nominal		
	Minimum		

4.2 Mechanisms for Tag Access

In the data-link layer, the commands for setting up the access and communication with one or more tags are defined. It also specifies the logical memory of a tag and how to access it.

4.2.1 Tag Memory

The memory is logically divided into four banks. The logical distribution of the four banks is shown in Figure 4.9. The **RESERVED** memory holds the kill and/or the access password. The **EPC** memory includes the EPC, which identifies the object to which the tag is attached. It also contains the StoredCRC, which is a CRC-16 value over the EPC

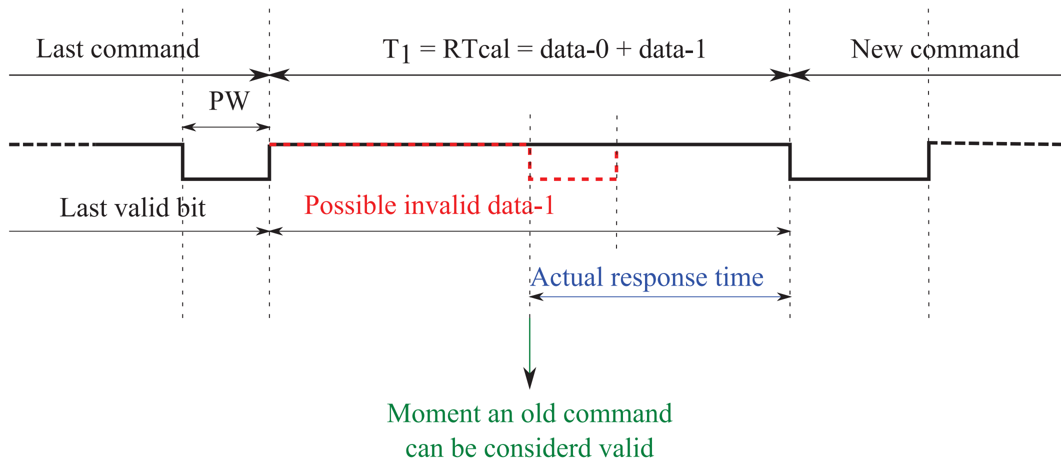

Figure 4.8: The actual time from the moment a command can be considered as valid to the moment the tag has to respond or be ready to receive a new command.

and the StoredPC. The StoredPC holds different parameters relating to the memory. The **TID** memory holds the ISO/IEC 15963 Class Identifier and other vendor-specific data. The **USER** memory holds user-specific data.

4.2.2 Inventoried Flag and Selected Flag

The communication between interrogator and tag is initiated with a *Query* command, which starts a new inventory round. A tag participates in an inventory round during one of four sessions. The *Query* command indicates in which session the inventory round is started. For every session a tag has an *Inventoried* flag, that could be set either to **A** or to **B**. It also has a *Selected* flag, that could be asserted (**SL**) or deasserted (∼**SL**). These flags are modified with the reception of a *Select* command with the proper parameter.

If a *Query* command starts a new inventory round in the same session as the previous one, then the tags that participated during the last inventory round, may invert their *Inventoried* flag for the particular session form A→B or B→A.

4.2.3 States of a Tag

A tag shall always be in one of 7 states. A tag shall also have a slot counter, which is set to a random value at a *Query* or *QueryAdjust* command. The slot counter is decremented at a *QueryRep* command.

The behavior of the tag and its response to a command depend on the current state and the value of the slot counter.

- **Ready state.** The **Ready** state is the initial state of a tag. Regardless of the previous state, a tag always returns into the **Ready** state after it loses power. The only exception is a killed tag, where the tag permanently remains in the **Killed** state.

  When receiving a *Query* command with parameters that apply to the current flags of the tag, a random value is loaded into the slot counter. If the value of the slot counter is non zero, the tag moves to the **Arbitrate** state. If the value is zero, a tag moves to the **Reply** state.

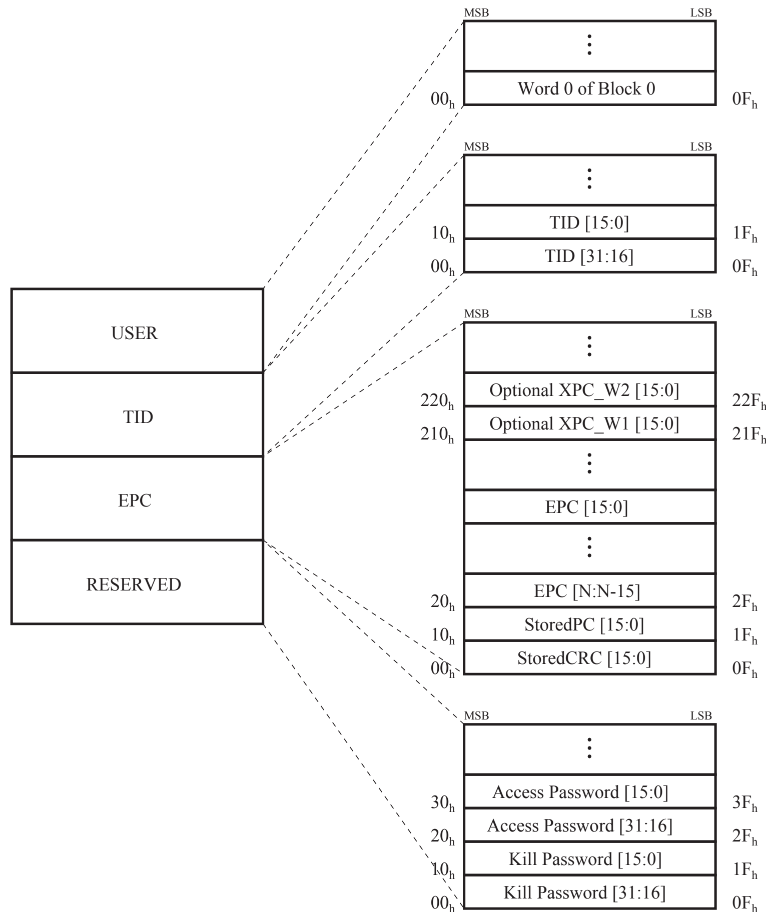

Figure 4.9: Logical distributions of the tag memory divided into four banks. Adapted from EPCglobal Inc. [11]

- **Arbitrate state.** Tags in the **Arbitrate** state participate in the current inventory round because their flags matched the parameters of the last *Query* command. However, the slot counter is still non-zero. If a tag receives a *QueryRep* or a *QueryAdjust* command that sets the slot counter zero, the tag transitions to the **Reply** state. On a *Select* command or a non-matching *Query* command, a tag returns to the **Ready** state.

- **Reply state.** In the **Reply** state a tag waits for a valid *ACK* command. If this is not received within a specified time, or an invalid *ACK* command is received, the tag transitions back to the **Arbitrate** state. On a valid *ACK* command the tag transitions to the **Acknowledged** state.

- **Acknowledged state.** If a tag does not receive a valid command within a certain time, it transitions back to the **Arbitrate** state. If a tag with a non-zero access password receives a valid *Req_RN* command, it transitions to the **Open** state. Otherwise, if the tags access password is zero, it transitions to the **Secured** state.

- **Open state.** In the **Open** state, an interrogator that transmits valid access commands to a tag has access to its memory. Only the *Lock* command is not executable

in the **Open** state. For the time between tag response and interrogator transmission (T_2 in Table 4.1), there are no constrains while the tag is in the **Open** state.

- **Secured state.** In the **Secured** state, an interrogator can access the tag and its memory with all access commands. For the time between tag response and interrogator transmission (T_2 in table4.1), there are no constrains while the tag is in the **Secured** state.
- **Killed state.** A tag in **Kill** state is disabled. It sends no answer to the interrogator. Only on the transition to the **Kill** state it responds to the interrogator, to confirm the transition. The tag remains in **Kill** state even after it loses the power supply.

4.2.4 Managing Tags

The management of tags is divided into three steps - *selecting*, *inventorying* and *accessing*.

For the *selecting* step there is only the *Select* command. With this command the tags, to which the interrogator desires access, are selected. The *Select* command can be repeated several times until all desired tags have been selected.

In the *inventorying* step an inventory round is started with a *Query* command in one of the four sessions. With additional commands a single tag can be identified and prepared for further access.

In the *accessing* step the tags and their memory are accessed with appropriate read and write commands. For the validation of some interrogator commands and some responses from a tag, the tag needs to do a cyclic redundancy check. A tag should be able to calculate both, a 16-bit CRC-16 value and a 5-bit CRC-5 value.

4.2.4.1 Selecting Step

With the *Select* command those tags are chosen, who shall participate in a subsequent inventory round. With the parameters *Target* and *Action* of the *Select* command the Selected flag and the four Inventoried flags can be set to a specific value. The parameters *MemBank*, *Pointer* and *Length* specify the memory location that shall be compared with the parameter *Mask*. All tags, that match the comparison, change their flags appropriately.

Since the *Select* command performs a read operation on large parts of the tags memory, the handling of this command requires comparatively many resources. In addition to that, the EPC Gen2 standard allows a bit pointer in the *Pointer* parameter. Thus the order of the data for the comparison against the tags memory is not byte wise aligned. This requires a reordering by shift operations to match the data against the byte-wise aligned EPC data. Also the *Mask* parameter must not have a data amount that is a multiple of 8 bit. For these reasons the handling of the *Select* command has a more complex implementation than for other commands.

4.2.4.2 Inventorying Step

The inventory round is started with a *Query* command. The parameters *DR*, *M* and *TRext* are relevant for the modulation of the tags answer. Depending on them, the tag is configured in a certain way for the specific inventory round. The *Session* parameter indicates in which session the inventory round is started. The parameters *Sel* and *Target* indicate which tags participate in the inventory round. The parameter *Q* is used to specify the number of digits for the random number, which is loaded into the tag's slot counter.

If a tag loads zero in the slot counter, it backscatters a 16-bit Random or Pseudo-Random Number (RN16) as answer to the interrogator. Otherwise, it does not respond.

On a *QueryRep* command, whose session parameter match the current inventory round, the tag decrements its slot counter. If the slot counter is zero, the tag responds with a RN16.

If a tag gets the *QueryAdjust* command, which session parameter matches the current inventory round, it loads a new random value into the slot counter. The *UpDn* parameter specifies if the number of digits for the new random number is increased or decreased, or if it remains the same. If a tag loads zero into the slot counter, it backscatters a RN16 to the interrogator as response.

When an interrogator gets a RN16 response, it confirms this with an *ACK* command. The tags response to the *ACK* command implies the full or a truncated EPC, depending on the *Truncate* parameter of the last *Select* commands in the previous select step. The truncated answer to an *ACK* command depends also on the *Pointer* and *Length* Parameter of the *SELECT* command. Since the *Pointer* parameter is a bit pointer and the *Length* Parameter is not byte-wise aligned, the EPC data that is backscattered has to be reordered by shift operations to be set into the right order for the transmission. Thus also like the *SELECT* command, the computation of the answer for the *ACK* command requires a more complex handling than for other commands.

After a valid *ACK* command the tag is in the **Acknowledged** state, and the access to the tags memory using access commands is possible. After a *NAK* command, a tag transitions back to the **Arbitrate** state without affecting its flags.

Since the purpose of those commands is to pick up one tag out of plenty, it requires them to be repeated several times by an interrogator until a single tag is selected. One can assume therefore, that those commands are designed with an accordingly short command length in comparison to the other commands. The design even abdicates a CRC checksum, except for the *Query* command, where a CRC-5 checksum is used. All other commands implement a CRC-16 checksum.

Nevertheless, if one of the “Query” commands leads to a slot counter equal to zero, this provokes different operations on the tag that have to be finished within the time T_1 in Table 4.1. Therefore these commands are seen as the most critical in this standard.

4.2.4.3 Accessing Step

Before accessing the tag’s memory with an access command, the reader sends a *Req_RN* command with the last received RN16. The tag responds to this with a new RN16, which is called *handle* subsequently. Then it transitions, depending on the access password, from the **Acknowledged** state either to the **Open** or to the **Secured** state. The *handle* will not be changed for the current inventory round and it is included in every subsequent access command as a parameter.

Read, *Write* and *Kill* commands are valid in the **Open** state as well as in the **Secured** state. Tags accept *Lock* commands only in the **Secured** state. For the *Write* and *Kill* command, the data that is sent to the tag, is obscured by an XOR operation with a random number. Therefore these commands must be preceded by a *Req_RN* command to get a new RN16.

The *Lock* command is used to lock and unlock the tags memory for access. An unlocked memory is accessible from both, the **Open** and the **Secured** state. A locked memory is only accessible from the **Secured** state. The memory can also be permanently locked

or unlocked. A memory that is permanently unlocked is accessible from both states and cannot be locked any more. If it is permanently locked, it is not accessible from any State.

A tag is killed in two steps by sending two *Kill* commands, each containing one half of the kill password. Before each of these commands a *Req_RN* command is required, because the kill password has to be obscured with a random number. If a tag is killed, it remains in the **Kill** state even after loss of power supply.

All commands of the *accessing* step have 16 bit long CRC-16 checksum at the end of the data stream. The last data before the CRC-16 value is the *handle*. Thus the commands are distinctly longer than the commands of the *inventorying* step. This in turn has the benefit that a tag has more time to handle the outcome of the command.

4.3 Possible Extensions and Improvements

It seems that the described protocol scheme and commands are primarily based on managing tags. One could assume that during the design of the standard, handling a bunch of tags was more important than security issues. Nevertheless the standard provides protocol extensions for custom commands. Also an EPC Gen2 tag has an user memory which can be used for custom data.

In Man et al. [19] and Ricci et al. [20] designs are given that implement a standard ISO 18000-6C baseband processor with additional security functionality. More precisely they have an additional AES functionality. Unfortunately in Ricci et al. [20] it is not stated how the AES functionality is harmonized with the EPC Gen2 standard.

As already mentioned in Chapter 3, a cryptographic primitive like the AES cipher needs a proper protocol to assure the requested security goals. Thus an implementation of a cryptographic primitive needs a certain flexibility in its applicability, to be usefully for arbitrary security requests on an RFID tag. In Man et al. [19] the suggested data flow encloses the protocol of the EPC Gen2 standard, which would make the tag incompatible with readers that do not implement an AES functionality.

Although, by providing an AES implementation, a custom processor tag with extended functionality gives a strong cryptographic primitive, it also could be limited in its use, if this primitive is restricted by its hardware implementation. The range of application of an EPC Gen2 tag with a security tool would increase, if on one hand the basic commands of the EPC Gen2 standard would still be applicable, and on the other hand, the additional functionality would be adaptable to a certain scenario. This leads to the idea of having a tag, based on a standard microcontroller instead of a custom processor, which is discussed in Chapter 5.

Chapter 5

Architecture

The idea of this work is to create a flexible implementation of an EPC Gen2 tag that allows easy modifications for further extensions. As already discussed in the previous chapters the EPC Gen2 standard does not provide appropriate security techniques. There are several proposals of EPC Gen2 tags that implement such security extensions. Nevertheless most of those implementations are restricted in their applicability. In this work an implementation is proposed that is based on a standard microcontroller. Thus the tag is fully programmable such that additional functionality should be easy to apply.

5.1 System Overview

Figure 5.1 gives an overview of the system. The tag consists of an analog unit and a digital unit. The analog unit of the front-end modulates and demodulates the carrier. It also creates the clock signal for the digital unit. The digital unit itself consists mainly of three further parts: the digital front-end, which is responsible for the command detection and the low level functionality, the microcontroller, which is responsible for the command handling and the higher functionality, and at last the memory, which contains the RAM, the program memory and the EEPROM.

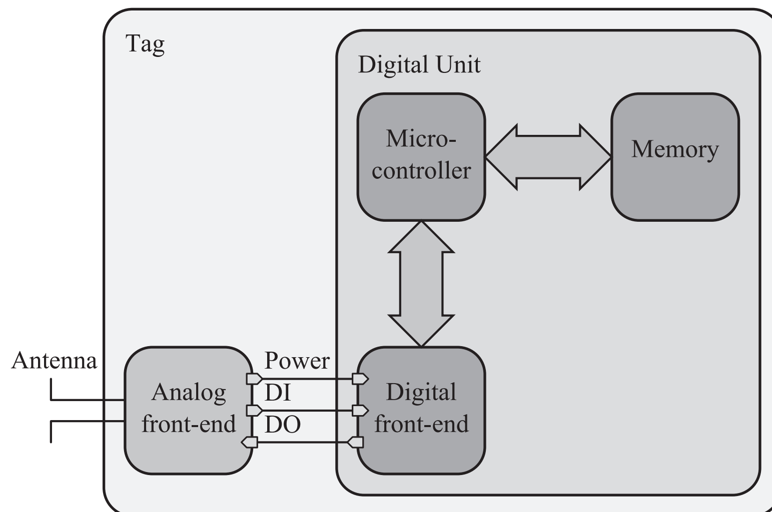

Figure 5.1: Main parts of the system

5.2 Trade-off between Hardware and Software

To ensure high flexibility the digital unit of the tag is based on a standard microcontroller. More precisely the microcontroller used in this work implements the widely used AVR instruction set of the Atmega128. This has the benefit of a lot of already existing tools available for use, like the GCC compiler (see Stallman and Community [28]). Moreover already optimized software for the AVR instruction set can be reused.

So in theory the additional hardware besides the microcontroller itself, only needs to act as an interface between the analog frontend and the microcontroller. The minimal functionality of this interface is to interchange the data between the microcontroller and the analog front-end and to provide the clock signal for the microcontroller. The front-end can be seen as a buffer between the serial and the parallel data flow.

Nevertheless handling mostly everything in software, would lead to large code. This means, that a lot of instructions would have to be processed, to achieve the desired functionality. Thus the minimal clock rate for the microcontroller needed to meet the timing constrains of the EPC Gen2 standard would increase. Since a higher clock rate leads to higher power consumption and low power is a basic requirement for a passively powered RFID tag, it should be considered to swap some of the functionality into the digital front-end. With some additional logic inside the digital front-end the required code could be reduced significantly. Thus, the reduced clock rate would potentially save more power than the additional hardware would produce. For the following parts in the protocol, swapping them into hardware would be reasonable:

- Operations that have to be applied on every transmitted or received bit could be done ad hoc. For example some commands in the EPC Gen2 standard require a Cyclic Redundancy Check (CRC). Either a received command has a checksum at its end, which has to be verified by the tag against the preceding part of the command, or a checksum has to be added to the end of a transmitted answer to a command. In both directions this CRC calculation could be handled online in hardware during the bit-wise transmission and reception of the data. This can be achieved by adding an appropriate shift register as suggested in EPCglobal Inc. [11].

  In software the calculation has to be done byte-wise instead of bit by bit, because the microcontroller processes data on an 8 bit architecture. With the reception of the last bit of a command the calculation and verification of the checksum would require more than one additional instruction. Even if the previously received data was precalculated to an intermediate value. Furthermore, it would cost additional instructions, between every received byte which would potentially lead to the necessity of a higher clock rate.

  If a shift register that is implemented in hardware logic is used, during reception the verified checksum would be signalized by a valid bit in a status register which has to be checked by the software. During transmission a control bit would trigger the adding of the checksum at the end of a response.

- Some low-level commands of the EPC Gen2 standard affect only the status and the configuration of a tag, but do not require a special handling of the transmitted data. These effects also depend on the current status and some of the received command parameters. Handling this issues in software would take numerous instructions, which operate on a bit level, just to update a few bits in a status register.

Additionally, those specific low-level commands of the EPC Gen2 standard have a comparatively short command length. The *QueryRep* command for example has only 4 bits. If such a command is handled in software, the microcontroller would not be able to precalculate the possible status outcomes, before the command turns out to be valid.

Furthermore, commands like *QueryRep*, do not affect the memory of the EPC Gen2 tag. According to the timing constraints such commands would require a higher clock rate at the end of a command to meet the time T_1[1] for the tag response.

Since such status and configuration updates require only a simple logic, it would be more effective to precalculate them by a hardware implementation in a temporary register. So the short time and thus the limited amount of instructions resulting with such short commands could be used for the recurring software parts that are applied on every received command. This is for example the validation of the received command itself. If in such a case the software indicates a valid command, a control bit could be set to signalize the hardware logic in the front-end to take over the status of temporary register into the real register.

- To reduce the clock rate of the microcontroller the operations for a tag response should start during the reception of the first data block. The first operation always is to check which command is possible to be received. This command detection takes additional instructions at the beginning of a command. Especially for the short commands, those additional instructions would represent a large part of the actual computation for the response itself. Thus for critical commands the detection of the command code should be done by hardware and signalized with a status flag by the digital front-end. In software the bits for those critical commands could then be polled, beginning with the most critical command.

- As mentioned in Chapter 4, a 16 bit random number is needed for the correct implementation of the EPC Gen2 standard. Thus a (pseudo) random number generator has to be implemented. In the beginning of a command reception it is not known, if the random number will be needed for the current response. Nevertheless a number hast to be ready for every command for the case that it is required if a certain command is detected. In software this could be handled in two ways. Either the calculation of the random number is triggered by a request for a proper response during the command handling, or it is precalculated before and/or during the actual command detection. In both cases additional instructions are needed, which would increase the clock rate of the microcontroller during the command handling.

  If a pseudo-random generator is implemented in hardware. The calculation of the random number could be done simultaneously to the command handling in an optimized hardware logic. The random number could then be directly accessible by the software through a register.

In respect of these points the digital front-end has a more functionality than just an interface. However, the high-level functionality can still be handled in software with the microcontroller.

[1]The time T_1 is discussed in Section 4.1.3

5.3 Design of the Front-end

The incoming and outgoing data into and from the digital front-end is shuffled bit-wise and modulated in a certain code format (see Chapter 4). Thus the front-end has to decompose the data when it is receiving and compose it into the proper format when it is transmitting the data. This functionality is achieved with the Rx-Tx-module in the front-end. The microcontroller used in this work has an 8-bit memory interface and an 8-bit datapath. Thus the central element for the data flow is a byte-queue with a first-in-first-out property (hereafter called FIFO), from which the data is pushed into and popped out of. Both operations can be done either by the microcontroller or by the Rx-Tx-module, depending on the direction of the data flow. During the reception the data is pushed into the queue by the Rx-Tx-module and popped out of the queue by the microcontroller. During the transmission it is the other way round.

Every part of a command or a response is processed through the FIFO. The only exception is the CRC, which is attached at the end of a tags response. The reason for this exception is that the CRC is calculated by an additional hardware module in the front-end and not by the microcontroller in software. When the last data is popped out of the FIFO, the CRC is passed directly from the CRC shift register to the transmitter module, which is responsible for the transmission.

The miscellaneous registers of the front-end are directly mapped into the IO-address-space of the microcontroller. So the transmitted and received data, but also the outsourced functionality, can be read and written by the `IN` and `OUT` instruction of the microcontroller. Figure 5.2 shows the main parts of the digital front-end. *DataWriteXDI*, *DataReadxDO* and *AddrxDI* are the signals from the I/O-bus of the microcontroller. *DemodxDI* and *ModDataOutxDO* are the serial input and output signals to the analog front-end.

5.3.1 Rx-Tx-module

The Rx-Tx-module is primarily responsible for the interaction with the analog front-end. This means that it has to detect an incoming command at the *DemodxDI* pin. This is done by a state machine that starts the command detection with the first rising edge on the *DemodxDI* pin. If the incoming signal has a valid format, the detected bit values are shifted into an 8-bit register bank. Every full byte is then pushed into the "First in, First out"-Queue (FIFO) until an error, in terms of an invalid format, or the end of a command is detected.

But there are also functionalities in the module, which are used to assist the software by analyzing and precalculating the data.

CRC-16 and CRC-5 validation and calculation. When the Rx-Tx-module is enabled to receive, with every bit shifted into the 8-bit register bank, the same bit is also shifted into a CRC-16 and a CRC-5 register bank. If a data stream with its valid checksum is shifted through those banks, the correspondent `crc5_valid` or `crc16_valid` bit is set to a logical one.

On the other hand, when transmitting is enabled, every bit shifted out of the 8-bit register bank is again shifted into the CRC-16 module. After the last bit of the response is shifted into the transmitter module, the CRC-16 register bank, holds the checksum for the response. This is also done during the startup of the tag. A special flag signalizes if the data pushed into the FIFO should be treated like data to be

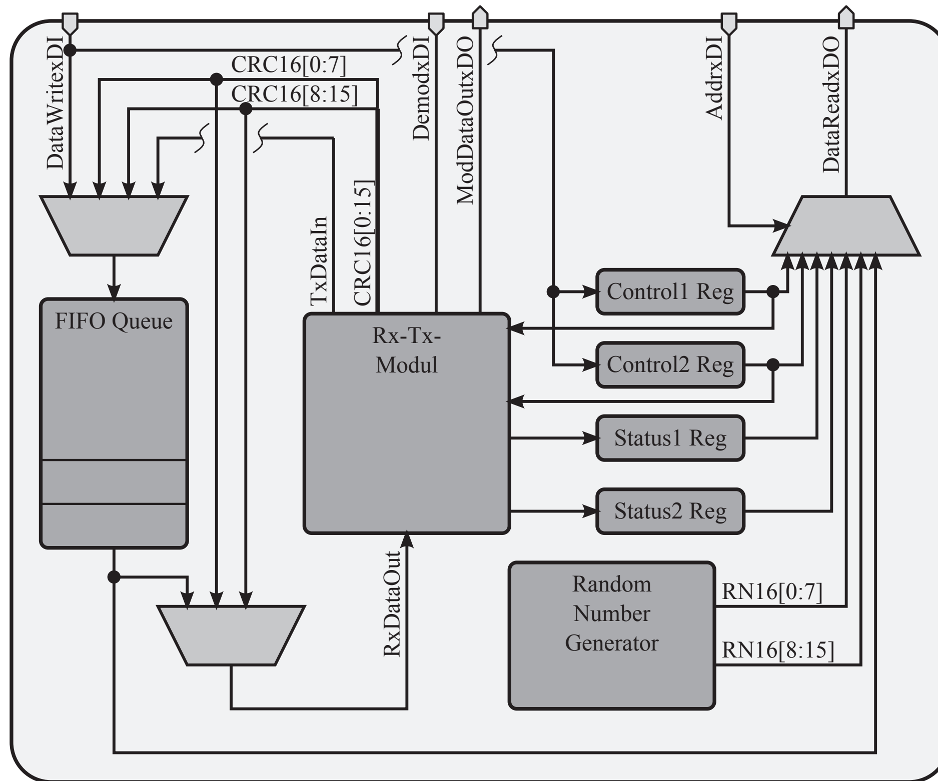

Figure 5.2: Overview of the digital front-end.

transmitted without the process of transmitting itself. This is used to calculated the StoredCRC value mentioned in Section 4.2.1.

Command detection. One task of the Rx-Tx-module that facilitates the command handling in software, is the command detection. For the time critical commands, the first incoming bits are checked against the command code of the critical commands and the correspondent status bit is set, if one of them is detected.

But also the incoming data is put into the right order, to save additional shift operations on the microcontroller. Therefore some bits of the command code of those critical commands are thrown away. This means that not all of the command code is pushed into the FIFO. This avoids the command parameters from being split into two different bytes, which would increase the number of instructions needed for the command handling in the software. Table 5.1 lists the time-critical commands and the bits that are thrown away. Figure 5.3 shows an example of how the parameters of the *Select* command are put into a proper byte order for easier handling.

Tag configuration. Some parameters of the *Query* command configure the timing of the tag response. Those configurations are stored in appropriate registers in the Rx-Tx-module. Since the *Query* command is a command that is detected by the Rx-Tx-module, the expected outcome for this configuration can be precalculated in hardware and stored into temporary registers. If a potential *Query* command turns out to be valid, the microcontroller signalizes with a control bit, that the values of the temporary registers shall be transfered into the current configuration register. Thus the Rx-Tx-module is immediately configured right for a following answer.

Table 5.1: Time-critical commands and the bits that are thrown away.

Command	Code	Original Length (bits)	Bits thrown away	New Length (bits)
Query	1000	22	1	21
QueryAdjust	1001	9	1	8
QueryRep	00	4	2	2
ACK	01	18	2	16
Select	1010	>44 (*Mask* is variable)	4	>40

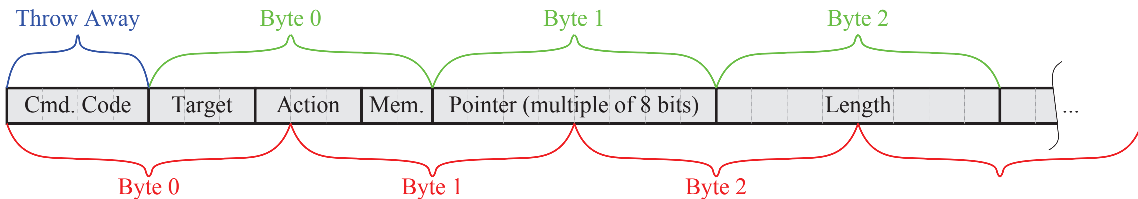

Figure 5.3: Throwing away the command code bits of the *Select* command to facilitates the handling of the parameters.

To reduce power consumption, the different modules are clocked and enabled only when a calculation step is needed. Additionally the parts that appear in both directions of communication, are implemented only once and reused for both ways. For example the CRC-16 register bank can be preset either for verifying the incoming serial stream, or for calculating the checksum of the outgoing stream. The logical operations are the same in both directions.

Since the provided clock from the analog front-end is processed through the Rx-Tx-module, the Rx-Tx-module implements a clock gating submodule that creates a variable clock for the microcontroller. In Section 5.4, how the different clock rates are applied through the individual stages of the tag communication is discussed. The incoming clock from the analog front-end is either divided by a constant, or a constant amount of clock pulses are gated through for every received or transmitted bit. Thus the clock signal depends on the data rate for reception and transmission.

5.3.2 FIFO

Even though the Rx-Tx-module and the microcontroller are coordinated through the control and status registers, they are not synchronous in terms of data exchange. Thus a First-In-First-Out queue is needed for the data transfer. Two pointers state the current position of the last and the first byte of the queue. With a pop operation the pointer for the first byte is increased. A push operation increases the pointer for the last byte. Additional signals give information if the queue is full, empty or if it contains just one byte. Thus the microcontroller can poll those signals to avoid reading from an empty and writing to a full queue.

The input port and the output port of the queue are directly mapped into the IO-address-space of the microcontroller at the same address. Thus an `OUT` instruction to this IO-address writes a byte to the pointer that points at the last byte of the queue (as long the queue is not full). An `IN` instruction from the same address reads the byte from the pointer that points at the first byte (as long the queue is not empty).

5.3.3 Pseudo Random Number Generator

For the 16-bit random number, required for the EPC Gen2 protocol, a Pseudo Random Number Generator (PRNG) has to be implemented. The PRNG is adapted from the proposal in Melia-Segui et al. [29]. It basically consists of an LFSR which has multiple feedback polynomials. The used polynomial depends on a true random bit that should be part of the analog front-end. The process of choosing a polynomial is implemented as a rotating wheel. If the random input bit is logical zero, the index for the current polynomial is increased by one. Otherwise it is increased by two. Figure 5.4 gives an overview over the PRNG. Table 5.2 shows the used feedback polynomials.

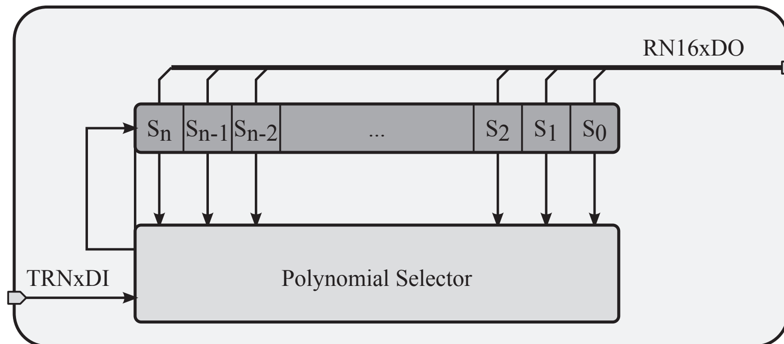

Figure 5.4: Basic structure of the PRNG (adapted from Melia-Segui et al. [29])

Table 5.2: 8 possible feedback polynomials are used for the PRNG

Feedback polynomials
$p_1(x)$: $1 + x + x^5 + x^6 + x^7 + x^{11} + x^{16}$
$p_2(x)$: $1 + x^4 + x^5 + x^6 + x^7 + x^{11} + x^{16}$
$p_3(x)$: $1 + x + x^3 + x^4 + x^5 + x^6 + x^7 + x^{11} + x^{16}$
$p_4(x)$: $1 + x^3 + x^5 + x^6 + x^{10} + x^{11} + x^{16}$
$p_5(x)$: $1 + x^5 + x^6 + x^{11} + x^{16}$
$p_6(x)$: $1 + x^5 + x^6 + x^{10} + x^{11} + x^{13} + x^{16}$
$p_7(x)$: $1 + x^4 + x^5 + x^6 + x^{10} + x^{11} + x^{16}$
$p_8(x)$: $1 + x + x^3 + x^4 + x^5 + x^6 + x^{10} + x^{11} + x^{16}$

5.4 Clock Frequency Constraints

The digital front-end is clocked at a maximal clock rate of 4MHz. This clock rate is chosen considering the minimal cycle time that is needed by the Rx-Tx-module to measure the timing constraints of the EPC Gen2 standard within the tolerance. Since the microcontroller consumes most of the power, it should be clocked at a lower frequency. For different commands and during the different stages of the communication, the required amount of operations within a certain time varies significantly. To optimize the power consumption, the microcontroller has to be clocked with an appropriate clock frequency, avoiding unnecessary idle operations. Thus the clock has to be adjusted in such a way so that the different operations and precalculations during the command handling on the microcontroller are split equally over time whilst a command is received. Therefore the

clock has not only to be switched on and off at certain points, but moreover it has to be dynamically adjustable.

The EPC Gen2 standard allows different data rates for receiving and also for transmitting. Further, during reception, a data-0 bit and a data-1 bit have different time lengths. Thus the clock should be independent of the current data rate and symbol. If so, with every received or transmitted bit the microcontroller gets only the amount of cycles that is needed to process the data.

On the basis of this approach the clock gating module, mentioned in Section 5.3.1, implements 4 different clock signals. Per default, the incoming clock from the analog front-end that is subsequently gated to the microcontroller is reduced by a constant factor and g. If reception is enabled, with every received bit from the front-end a certain amount of cycles is gated to the microcontroller. This is the so called Number of Cycles per Received Bit (RX-BIT-RATE). Analog, if transmission is ongoing, the microcontroller is cycled with the so called Number of Cycles per Transmitted Bit (TX-BIT-RATE), every time a bit is transmitted by the front-end. Therefore the microcontroller is not clocked with a constant rate, but rather it depends on the chosen data rates. The last option is to gate the incoming undivided clock itself. The microcontroller can enable the undivided clock via a control bit. This bit should be set at the end of a received command before the reception is finished, when the microcontroller has to validate the command in time for the response. Further this undivided clock will be stated as HIGH-CLOCK. Because not every command needs the same amount of clock cycles to calculate the answer, this bit would be cleared by the microcontroller before the actual transmission starts. Hence the clock frequency will be reduced to the default frequency during the last part of the time between reception and transmission. The amount of clock cycles that are gated during the time T_1 (see Section 4.1.3) hereafter is called TX-WAIT-CYCLES. In Section 6.4.2 how the TX-WAIT-CYCLES give a boundary for the incoming clock frequency and how they depend on the RX-BIT-RATE and the TX-BIT-RATE is discussed. A typical clock rate distribution looks like Figure 5.5. For the majority of the commands during a typical communication the HIGH-CLOCK will be just a peak in the power consumption.

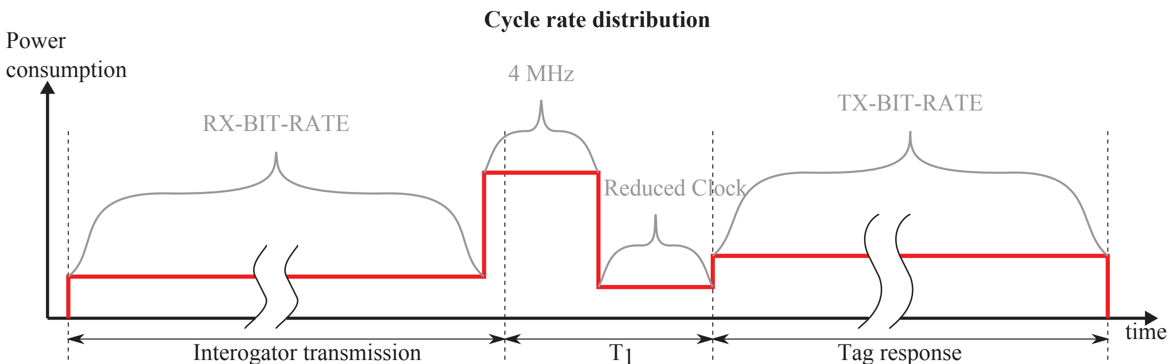

Figure 5.5: Typical power distribution over the time. The highest clock rate will resolve during the time T_1 in Table 4.1.

5.5 Optimization Approaches in Software

The source code for the microcontroller was thought to be written in C. With the highest optimization level `-o3` of the GCC compiler, the source for the critical commands was still not fast enough. This means that the microcontroller needed an inadmissible amount of instructions to process the received commands. Thus the critical parts of the software have to be optimized manually by self-written assembly. The following approaches are applied to optimize the speed of the source code:

Effective register use. A possible optimization is to avoid operations on the stack. Because of the precalculations, while receiving a command, many temporary and intermediate values arise. Even with the best optimization the compiler cannot arrange the temporary results in a way that they remain in a register. Instead, the values are saved on the stack, which brings additional instructions for moving the temporary values to and from the stack.

For this kind of optimization no real optimization technique or method is required, but the manual reordering of the intermediate values for an effective register use can save many additional instruction cycles by avoiding the use of the stack. For an example how instruction cycles can be reduced see Listing 5.1 for an inefficient way to handle an intermediate value, done by the compiler. When an intermediate 16-bit integer value that is stored on the stack has to be manipulated, the value has to be popped out of the stack first and afterwards pushed back into it. The `ldd` and the `std` instructions that apply these stack operations need two cycles. Thus every time the value is manipulated, 8 additional instructions cycles are needed, for saving the temporary value on the stack.

Listing 5.1: Inefficient stack operation

```
1       //temporary value is stored on stack
2       //at stackpointer+23
3
4       ldd       r26, Y+23
5       ldd       r27, Y+24
6       add       r26, r20
7       addc      r27, r21
8       std       Y+23, r26
9       std       Y+24, r27
```

It is not enough to write inline assembly for reordering the registers only in parts of a computation. The optimized part, written in inline assembly, is still surrounded by C code. This has the effect that at the beginning and at the end of the self-written assembler there are still stack operations. Thus, it is necessary to write the whole part of the handling for a specific command in assembly. With such large code parts, it is a challenge to manually reorder the instructions in a way that the temporary variables remain in registers without using the stack.

Avoiding subroutines. Since a `call` (Subroutine call) and a `ret` (Subroutine return) instruction need 4 cycles, the use of subroutines should be avoided. Therefore the implementation for the whole command handling is written in one main function, besides the subroutines for the EEPROM memory access. This not only saves the cycles for the `call` and `ret` instructions, but also avoids additional register usage, since, according to the convention of the compiler, the arguments of a subroutine are

passed through by registers. On the other hand, this approach has the disadvantage, that the code size increases, because reoccurring code parts are not reused.

Avoiding jumps. The `jmp` (Jump) and the `rjmp` (Relative jump) instructions need 2 and 3 cycles and thus they also produce many additional cycles. Especially in combination with conditional jumps and branches, reordering of the code can save unnecessary cycles.

Data type optimization. In some parts of the code 16-bit variables are used. Operations on these variables are always applied on both bytes. In some cases, only the low or the high byte is important, but the compiler still handles the whole 16-bit variable. Thus the instructions on the other byte can be saved. An example for a data type optimization is shown in Listing 5.2 and Listing 5.3.

Listing 5.2: Operation on a 16-bit variable

```
1      //lock_lookup_kill += (tmp8_1&0x30)>>4
2
3      mov     r20, r26
4      ldi     r21, 0x00
5      andi    r20, 0x30
6      andi    r21, 0x00
7      asr     r21
8      ror     r20
9      asr     r21
10     ror     r20
11     asr     r21
12     ror     r20
13     asr     r21
14     ror     r20
15     add     r24, r20
16     adc     r25, r21
```

In Listing 5.3 the upper 4 bit of the 8-bit variable `tmp8_1` in register `r26` have to be added to the 16-bit variable `lock_lookup_kill` in the registers `r24` and `r25`. Listing 5.3 shows an optimized version, where only 8 bit are processed instead of 16 bit.

Listing 5.3: Optimized operation on a 16-bit variable

```
1      //lock_lookup_kill += (tmp8_1&0x30)>>4
2
3      mov     r20, r26
4      andi    r20, 0x30
5      swap    r20
6      add     r24, r20
```

Listing 5.3 also shows a typical optimization that can be applied by analyzing the outcome of the desired result. Since the `andi` instruction in line 3 has a result looking like $\texttt{00xx0000}_b$ instead of the 4 `asr` (arithmetical shift right) instructions, one `swap` (swap nibbles) instruction also brings the desired result ($\texttt{000000xx}_b$).

Loop unrolling As already mentioned, the EPC Gen2 standard allows bit-pointers and data lengths that are not a multiple of a byte. This leads, during the reception of a *Select* command and also during the transmission of a truncated reply to an *Ack* command, to the same bottleneck. Possibly large amounts of data have to be

shifted into the right order, depending on the bit-pointer. The AVR instruction set of the Atmega128 has only bit-wise shift instruction, namely the `lsl` and the `lsr` instruction. Since the number of shifts depends on the *Pointer* parameter of the *Select* command, the shift instructions cannot be done one after another. A loop, depending on the bit-pointer, has to be used instead. This is shown in Listing 5.4.

Listing 5.4: Variable shift operation[1]

```
1        mov      r31,r26 //move bit pointer in temporary register
2        rjmp     2f      //relative jump to local label 2 forward
3  1:    lsr      r25     //register that has to be shifted
4  2:    dec      r31
5        brpl     1b      //relative jump to local label 1 backward
```

This way one shift operation needs 4 instruction cycles (`brpl` needs two cycles if condition is true). Worst case the data has to be shifted by 7, so 28 cycles are needed just for bringing the data into the right order. This can be reduced by writing extra code for all 8 possibilities the data has to be shifted depending on the bit-pointer. With some additional optimization the size of this particular code increases then by a little more than 7 times. However, in the worst case, only 5 instructions are needed for shifting (1 `swap`, 1 `andi` and 3 `lsl` or `lsr` instructions). Listing 5.5 shows an outtake of the extra code to handle the different possible values of the bit-pointer. Thex in the labels `BPx` in Listing 5.5 represents the number of the shift instructions that are done by the specific code part.

Listing 5.5: Fixed shift operation

```
1  BP3:  lsr      r25
2        lsr      r25
3        lsr      r25
4        rjmp     END
5  BP4:  swap     r25
6        andi     r25,lo8(15)
7        rjmp     END
8        ...
9
10 BP7:  swap     r25
11       andi     r25,lo8(15)
12       lsr      r25
13       lsr      r25
14       lsr      r25
15 END:  ...
```

Most optimizations not only reduce the required instructions for a certain functionality, but also decrease the code size. But some approaches like loop unrolling increase the code size considerably. In Section 6.4.1 the optimization approaches on the final code are evaluated regarding the clock cycle requirement and the code size.

5.6 Control flow between Hardware and Software

The microcontroller works independent of the current front-end status. To assure the correct access to the front-end and subsequently the right processing of the Rx-Tx-module,

[1] `2f` and `1b` in Listing 5.4 are a so called local label. They denote a jump forward to the next label `2` and a jump backward to the next label `1`

various status and control flags are used. These flags are represented by two status byte registers and two control byte registers. Like the input and output of the FIFO, those byte registers are directly mapped into the IO-address-space of the microcontroller.

Despite the fact, that the microcontroller and the front-end are independent, they still have to synchronize with each other. Synchronization here is not meant in terms of processing the same data at the same time, but in terms of what both expect of each other. A summary over the registers and their flags is given in Appendix B

5.6.1 Controlling the Front-end

The front-end starts clocking the microcontroller when it is powered by a UHF-signal and thus it is receiving a command. However, the microcontroller decides if the Rx-Tx-module of the front-end is enabled to process the command and subsequently push it into the FIFO. On the other side, when the microcontroller pushes data into the FIFO, the front-end has to be enabled for transmitting and ignoring further incoming signals. Therefore the most essential control flags are the `rx_enable` and the `tx_enable` flags.

The data, which is processed by the microcontroller for transmission is seen as a high level data. This means, that the data that is pushed into the FIFO does not include protocol specific low level information. This additional low level information like header bits and CRC checksums is handled by the front-end in hardware logic. If additional data has to be preceded or appended by the front-end, the specific control flags have to be set in the software, to activate the accordant modules for those functions.

When the front-end detects a potential command that would change the configuration for the communication, it stores the potential parameter into a temporary register. The validation of the correctness of the command is done by the microcontroller, thus it has to set a flag that stores these configuration registers.

To meet the timing constraints for the command response, the microcontroller has two options to speed up its command handling. One possibility is to raise the clock frequency from the front-end. If the according flag is set, the highest clock is enabled, independent of the front-end status. Another option is to force the front-end to push the incoming data earlier into the FIFO. With the `push_after_size` control bits, the microcontroller triggers after how many received bits, the front-end has to push the temporarily stored data from the serial input of the analog front-end into the FIFO. Thus it can process the incoming data earlier.

After a received command, the microcontroller has to decide whether to give a response, or not. The Rx-Tx-module of the front-end internally starts a counter to measure the time T_1, before it starts with the preamble of the response, or it transitions into idling. If this time is going to be exceeded, because of a memory write operation on the tag, this has to be signalized by an additional flag, to avoid the transition into idle. The response itself is pushed byte-wise into the FIFO. With the last byte certain flags have to be set accordingly, to signalize the end of the response.

5.6.2 Sensing the front-end

The microcontroller uses various flags to be aware of the current state of processing of the front-end. Some of them have to be polled to initiate a certain operation, like the `cmd_detected` flag, which starts the command handling. Others are just checked at a certain point, like the `crc16_valid` flag, which verifies the checksum. Detailed information about the individual status flags is given in Appendix B

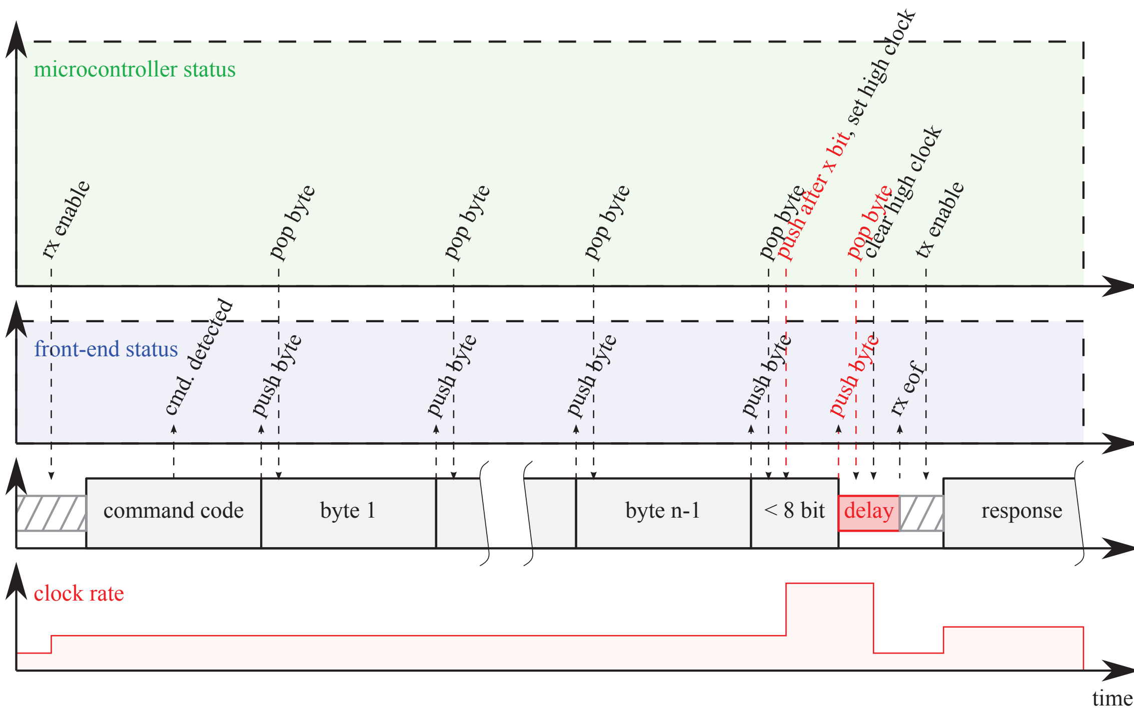

Figure 5.6: Example control flow with avoiding the delay by using the `push_after_size` bits.

5.6.3 Critical points of operation

As discussed in Section 4.1.3, there is a certain delay until the `rx_eof` flag can be set by the front-end. To avoid this delay, the `push_after_size` bits are used. When the microcontroller assumes that for a valid command only, e.g., two bits are left, it sets the `push_after_size` bits to the according values. Thus the front-end pushes the last data earlier into the FIFO and the microcontroller continue its processing as if it was verified that they were the last bits. So when the current `rx_eof` flag is set, it has only to check the `last_data_invalid` and the `fifo_not_empty` flag, to verify the correctness of the command. Figure 5.6 shows an example of the control flow with avoiding the delay by using the `push_after_size` bits. Figure 5.7 shows an example of the control flow without avoiding the delay. In Figure 5.7 one can see, that with the delay the time constrain can not be fulfilled, although the high clock is present for a longer time.

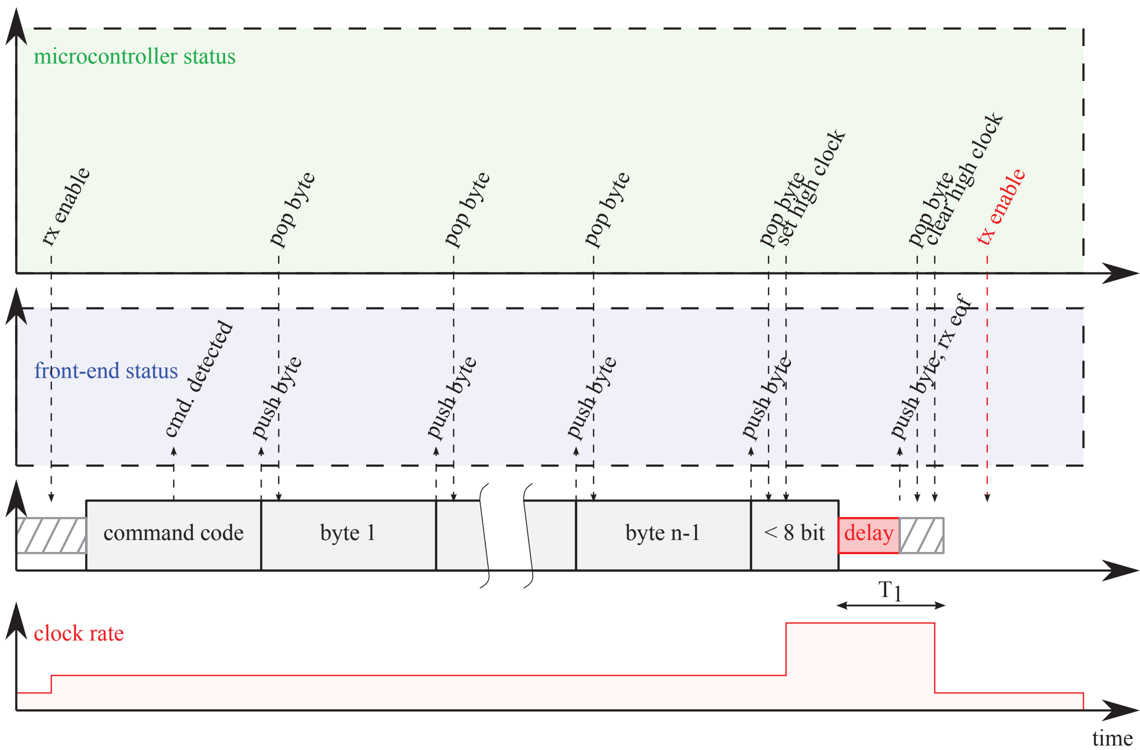

Figure 5.7: Example control flow with delay. The time constraints cannot be fulfilled.

Chapter 6

Implementation and Evaluation

In this design a standard microcontroller is expanded with a digital front-end for RFID communications, according to the EPC Gen2 standard. This front-end is not only used to execute the communication process instructed by the software from the microcontroller, but the frontend has also to support the software in terms of reducing the executed instructions on the microcontroller. Therefore it implements the functionality for time-critical parts of the EPC Gen2 standard, but also certain functionalities that are reoccurring in most of the commands and that could be parallelized during the command specific code execution. Moreover the front-end can regulate the clock rate of the microcontroller to the minimal required amount, depending on the executed command.

Therefore the front-end and the software on the microcontroller have to be able to synchronize their execution progress at different points of the communication process. In turn, this requires an adequate interface and a harmonized software.

For developing and verifying such a design and moreover the interface between the particular parts, exhaustive tests are necessary. Thus a software model has been written that simulates the microcontroller and the front-end. This model is able to execute the source code for the microcontroller. Having such a model makes it possible to apply a bulky test of the communication process on one hand, and on the other hand the usage of the model is an easy way to adapt the concept of the front-end and the interface, if it is required. Additionally with this model it is easier to design and verify new commands, which extend the protocol of the EPC Gen2 standard.

During the implementation, the software model is used for different steps of development:

1. Verifying the functionality of the software for the microcontroller.

2. Verifying the design of the front-end and the interface to the microcontroller.

3. Optimizing the software for the microcontroller.

4. Evaluating the minimal clock rates at different points of the RFID communications for different commands of the EPC Gen2 standard.

5. Testing and verifying possible extensions.

6.1 Simulated Model of the System

To be able to simulate the concept and to develop the code for the microcontroller a model has to be written. To have a uniform design flow for this model it is important that the same test cases can be applied to it, as on the actual hardware design.

This model reuses the design of the simulator SimulAVR (for detailed information about SimulAVR see Roth et al. [30]). SimulAVR is a simulator for a different AVR microcontroller. The source code is published under GPLv3 [31] license and freely available.

The SimulAVR simulator is designed in a way that it is possible to write own I/O-modules and add them to the I/O-interface of the microcontroller. Such I/O-modules are represented by routines, that implement the functionality of the modules. These routines read from and write to the I/O-bus of the microcontroller and hence communicate with it.

Nevertheless, the design of the simulator has the limitation that the routine of an I/O-module is executed only once for every simulated cycle. Thus, it is not possible to clock the I/O-module at a higher clock rate than the microcontroller. Further, and more important for the purpose of this project is that in the simulation the front-end module can not be run at a constant clock rate and thus cycle the microcontroller dynamically, as it is the intention for the physical design. During the simulation, the microcontroller and the I/O-modules run at a fixed and constant clock rate, which does not reflect the final system in this project.

Nevertheless this problem can be handled in a way so that it is not relevant for the results of the simulation. In the next section, how this can be managed is discussed.

6.2 Test Case Generation and Appliance

The idea for testing the software model is to use the same test scripts that are used for the Hardware Description Language (HDL) simulation, so the same test cases can be applied on both models. The HDL model is simulated with so called TCL-scripts. These TCL-scripts basically operate on the I/O pins of an HDL model. A typical script specifies at which time an input signal is set to a certain value, and checks the expected values on the output signals. To make clear and readable test cases, various routines are designed to compose the sequence of the script. This routines are invoked by an overlaying script that can be seen as a higher level script, which only specifies the input sequences and the expected output, without considering the timing values. The timing values are specified through some configuration values at the beginning of the high level script.

This high level script is also reused for the simulation of the software model. Here the model of the front-end reads the script and out of its content the model sets its intern status and the interface to the microcontroller. But instead of applying the input values at a certain time, the front-end model has to interpret the script in a different way. The reason for this is that in the simulator it is not possible to have a different clock rate for the microcontroller and the I/O-modules. Instead, the front-end applies the input values after a certain amount of executed cycles, depending on the clock rate that would result out of the configuration TCL-script, and depending on the values for the RX-BIT-RATE and the TX-BIT-RATE[1]. Hence this model does not simulate the real time, because the microcontroller is clocked with a constant and fixed cycle rate.

[1]RX-BIT-RATE and TX-BIT-RATE are specified in Section 5.4

Moreover, while running at the same clock rate, the front-end counts the cycles until a certain event is triggered by the microcontroller. The result of this counting is logged in a file that can be used to evaluate the required cycles for a certain part of the execution.

For applying a bulk of different test cases, a C-routine is written, which automatically creates large TCL-scripts. For different scenarios that should be tested, the parameters of a command are varied and a script is automatically generated, depending on the expected result of the command. For these test cases the different random numbers in the EPC Gen2 protocol are assumed as fixed values, since the behavior of the system should be predictable.

6.3 Extending the Protocol

Since the tag in this work is designed to be easily expandable in its functionality, adding a custom command requires minor effort. Actually, only an additional code is needed to handle a new command, since the low level functionality for handling a command should be handled by the front-end, like for the other commands.

The EPC Gen2 standard specifies all command codes with a length of 8 bits that are not used in the current standard as reserved for further use. Thus a custom command shall have command code with a length of 16 bits. Also, like the other commands in the *accessing* step (see Section 4.2.4.3, the new custom command shall have a CRC-16 checksum attached after the actual command. It also shall include the *handle* value mentioned in Section 4.2.4.3.

A custom command complying with those conditions is not supposed to be at risk of exceeding the timing constraints of the protocol. Thus it is likely that the extension of the current source code that handles the various commands will not need optimization in assembler, as it is the case for the basic commands of the EPC Gen2 protocol.

As extension of the functionality, a simple authentication scheme is chosen. It shall be based on the AES-128 as cryptographic primitive. Moreover the necessary cryptographic computations are done in software. According to the working draft ISO/IEC WD 29167-10 ISO/IEC WD 29167-10 [32] the authentication command sent by the interrogator has an 80-bit long challenge and some other parameters. The tag shall add to these 80-bit and 16-bit long constant value and a 32-bit long random value, the so called *seed*. Together this data builds a 128-bit long block, which is the input for the AES-128 cipher. The response to this authentication command shall be the ciphertext output of the AES-128 cipher. The encryption key is stored in the user memory of the EPC tag.

To re-use the source code for the AES-128 cipher, the current *seed* is calculated also using the AES-128 cipher. Therefore an initial 128-bit (truly random and secret) seed value shall be saved in the user memory of the tag. Every subsequent seed value is computed by taking the current seed value as input for the cipher. The output of the cipher overwrites the current value. Only the first 16 bits shall be used as a seed for the authentication command.

Since the AES-128 computation takes a certain time, by means of cycles on the microcontroller, the response can not be calculated during the time T_1. However, the tag does not need to respond to within T_1, since the custom authentication command writes to the EEPROM for saving the seed.

6.4 Evaluation of the Software

The code that should be executed on the microcontroller to handle the commands according to the EPC Gen2 protocol in the first step is written in the programming language C. This first version is used for verifying the correctness of the intended code but also for verifying the design of the front-end and its interface. In the second step, after the verification, the code has to be optimized concerning the optimization approaches in Section 5.5. Together with the parameters for the dynamically adjustable clock frequency during the execution, the required instruction cycles for certain parts of the final code give boundaries for the minimal clock frequency, which is discussed in Section 6.4.2.

6.4.1 Optimization Results

Like mentioned in Section 5.3.1 the critical commands are the *Select*, *Query*, *QueryRep*, *QueryAdjust* and *Ack* command. To achieve higher optimization and faster code, it is required to implement those commands in assembly. One side effect of the elementary optimizations, besides having a faster code in terms of less executed instructions, is that the code size is also reduced. But at some point, with applying more optimization approaches, the source code gets less dynamic. This means that certain code parts are processed with less probability. Thus the code size gets larger again.

Table 6.1: Relation between the code size and the required cycles for the *Select* command, depending on the code optimization.

Code Optimization	Relative Code Size[1]	RX-BIT-RATE	additional cycles required
Optimization step 1	±0 bytes	6	4,639
		5	4,933
Optimization step 2	−502 bytes	6	1,213
		5	1,482
Optimization step 3	−4 bytes	6	114
		5	321
Optimization step 4	+602 bytes	6	75
		5	289

This effect is very significant for the *Select* and *Ack* command. As mentioned in Section 4.2.4.2 and Section 4.2.4.1, the handling for these two commands requires a reordering of the data through shift operations. For this special handling the approach of loop unrolling mentioned in Section 5.5 has the largest room for improvement. The relation between the various optimization steps, the code size and the cycle counts needed

[1]Relative Code Size means the difference in the code size in comparison to optimization step 1

to handle these two commands is shown, for a fixed RX-BIT-RATE, in Table 6.1 and Table 6.2.

For the evaluation of the *Select* command the microcontroller is clocked with a fixed RX-BIT-RATE. The required cycles are worst case values, determined through exhaustive test cases. In Table 6.1 the following optimization steps are applied:

- *Optimization Step 1*: The *Select* command is written in C and compiled with the optimization level `-o3`.
- *Optimization Step 2*: The *Select* command is written in assembly. All shift operations that depend on the *Pointer* parameter are handled in unoptimized loops.
- *Optimization Step 3*: The *Select* command is written in assembly. In Repeatedly executed code the shift operations that depend on the *Pointer* parameter are unrolled and handled for each case of the bit pointer separately.
- *Optimization Step 4*: The *Select* command is written in assembly. All shift operations that depend on the *Pointer* parameter are unrolled and handled for each case of the bit pointer separately.

Table 6.2: Relation between the code size and the required cycles for the *ACK* command, depending on the code optimization.

Code Optimization	**Relative Code Size**	**Required TX-BIT-RATE**
Optimization Step 1	±0 bytes	28
Optimization Step 2	− 340 bytes	16
Optimization Step 3	+264 bytes	5

For the evaluation of the *Ack* command in Table 6.2 the microcontroller is clocked with a fixed RX-BIT-RATE, while the TX-BIT-RATE is varied until the smallest value for a successful communication is found. The required cycles are worst case values, determined through exhaustive test cases. The following optimization steps are applied:

- *Optimization Step 1*: The *ACK* command is written in C and compiled with the optimization level `-o3`.
- *Optimization Step 2*: The *ACK* command is written in assembly. All shift operations that depend on the *Pointer* parameter are handled in unoptimized loops.
- *Optimization Step 3*: The *ACK* command is written in assembly. All shift operations that depend on the *Pointer* parameter are unrolled and handled for each case of the bit pointer separately.

6.4.2 Clock Requirement

The gain of the optimization measures is reflected in the reduction of the highest required clock frequency. As the front-end gates different clock rates to the microcontroller during the different stages of the communication process (see Section 4.1.3), the parameters for these different clock rates have to be evaluated against each other, to find the boundaries for the maximal required clock frequency.

Code optimization on the handling of the different commands on the microcontroller have effects on particular parts of the communication process. Thus it is necessary to distinguish the limiting commands for each parameter of the specific clock rates of the communication process.

The first version of the code for the microcontroller has shown that for the TX-BIT-RATE during the transmission of the *ACK* command is a bottleneck. Generally it is not reasonable to precalculate the whole answer to a command before the transmission starts. Moreover only the first byte that has to be transmitted is precalculated before `tx_enable` is set. Every other byte can be calculated during the transmission of the previous one. This approach has to be considered especially for the *READ* and the *ACK* command, since they can have accordingly long answers. The answer to an *ACK* command can be truncated. If this is the case, the data, which is read byte wise from the memory, has to be shifted in the right order depending on the bit pointer. Thus the answer for the *ACK* command requires more complex execution than for the *READ* command. In the final version of the code the *ACK* command needs 5 cycles per transmitted bit to precalculate and push the subsequent byte into the FIFO.

Table 6.3: Dependency between TX-WAIT-CYCLES and the RX-BIT-RATE for the *Select* command. The required cycles are the worst case values, determined through exhaustive test cases.

RX-BIT-RATE	**TX-WAIT-CYCLES**
3	866
4	570
5	282
6	75
7	49
8	38
9	35
10	35

The limiting command for the RX-BIT-RATE during the reception turns out to be the *SELECT* command, which has a variable command length, and potentially long command parameters. The *WRITE* and *READ* command have also a variable command length, but it is not expected to be as long as for the *SELECT* command, since for these two commands only the *WordPtr* parameter can vary in its length. Further the *SELECT* command has to also execute the already mentioned shift operations, to bring the received data into the right order, depending on the bit-pointer.

Since not every piece of information for the command handling can be precalculated at the time the reception is finished, the microcontroller needs a certain amount of instructions after the last data is received. In Section 5.4 this amount was stated as TX-WAIT-CYCLES. For the *SELECT* command varying the RX-BIT-RATE has a distinct influence on the required TX-WAIT-CYCLES, which is shown in Table 6.3 for the final version of the code. At a certain amount for the RX-BIT-RATE the TX-WAIT-CYCLES cannot be reduced any more.

Although the *Query* command has a fixed length it is also a limiting command for the RX-BIT-RATE and the TX-WAIT-CYCLES, since it causes many operations, depending on the current status. The dependency between those two factors for the *Query* command is shown in table 6.4

Table 6.4: Dependency between TX-WAIT-CYCLES and the RX-BIT-RATE for the *Query* command. The required cycles are worst case values, determined through exhaustive test cases.

RX-BIT-RATE	TX-WAIT-CYCLES
2	99
3	93
4	90
5	82
6	79
7	71
8	67
9	64
10	56
11	53
12	50
13	42
14	39
15	38
16	38

To find a trade-off between the RX-BIT-RATE and the TX-WAIT-CYCLES, a limit for one of these two values has to be determined. Therefore the *QueryAdj* and the *QueryRep* commands have to be analyzed. This is because during the reception both of these commands are forwarded to the microcontroller within one byte in the FIFO. Thus the microcontroller cannot do any precalculations during the reception. The calculations required for these commands are independent on the RX-BIT-RATE. All of the received data is processed after the reception of the command is finished. Table 6.5 and Table 6.6 give an overview on how many cycles are required to handle these commands depending on the state of the tag. These tables show that the *QueryAdjust* command needs 74 cycles after the last bit is received in the worst case. Thus when enabling the HIGH-CLOCK (see Section 5.4) at the end of the reception, the microcontroller has to get at least 74 clock cycles until the time T_1 is finished (see Section 4.1.3). This 74 cycles would correspond to the TX-WAIT-CYCLES. Since the TX-WAIT-CYCLES are fixed for every command, the value of 74 cycles can be used as boundary for the other commands to find the trade-off between the TX-WAIT-CYCLES and the RX-BIT-RATE. In Figure 6.1 one can see that considering the possibility of 74 TX-WAIT-CYCLES after the last data is received, for the limiting commands the required RX-BIT-RATE is 7 cycles per bit.

The required amount of TX-WAIT-CYCLES gives the value for the required clock rate, when the HIGH-CLOCK is enabled. Since the *QueryAdj* command limits the TX-WAIT-CYCLES to 74, the value for the HIGH-CLOCK can be calculated out of it. Having in mind that the lowest possible value for T_1 is 15.63µs, the required clock rate for meeting the timing constraints for all possible configurations is calculated as follows:

$$f_{Max} = \frac{74}{15.63\mu s} = 4.7MHz \tag{6.1}$$

As mentioned in Section 5.4 the front-end is designed to be clocked at a cycle rate of

Table 6.5: Required TX-WAIT-CYCLES for the *QueryAdjust* command dependent on the different internal values of the microcontroller and the command parameters. **NOTE:** Every RN16 is assumed to be 1234_h.

Current state	<u>Session</u> parameter does match	<u>UpDn</u> parameter	Current Q value	new slot value	required TX-WAIT-CYCLES
all	no	all	all	-	18
Ready	yes	all	all	-	29
Arbitrate	yes	000_b	6	0034_h	70
Arbitrate	yes	011_b	6	0014_h	74
Arbitrate	yes	110_b	6	0034_h	71
Arbitrate	yes	111_b	6	-	39
Arbitrate	yes	011_b	1	0000_h	71
Arbitrate	yes	011_b	0	0000_h	71
Acknowledged	yes	all	all	-	43
Open, **Secured**	yes	all	all	-	50

Table 6.6: Required TX-WAIT-CYCLES for the *QueryRep*.

Current state	<u>Session</u> parameter does match	resulting slot value	required TX-WAIT-CYCLES
all	no	-	22
Ready	yes	-	31
Arbitrate	yes	$>0000_h$	51
Arbitrate	yes	$=0000_h$	47
Reply	yes	-	28
Acknowledged	yes	-	42
Open, **Secured**	yes	-	51

4 MHz[1]. Thus the minimal value for T_1 that can be met is given as follows:

$$T_1 = \frac{74}{4MHz} = 18,5\mu s \tag{6.2}$$

The time T_1 depends directly on two factors of the communication configuration, namely T_{pri} (and thus *BLF*) and **RTcal**. Further those two factors partially depend on the same configuration values. Thus the minimal values for T_{pri} and **RTcal** still can be achieved, but not with all possible combinations. For example the maximal BLF of 640kHz (and thus the minimal T_{pri}) can still be combined together with the lowest possible *Tari* value, but therefore the **RTcal** value has to be $3 \times$ *Tari* (cf. Figure 4.2)

[1]For higher cycle rates, a few constants in the design have to be adapted for the front-end to measure the timings respectively to the cycle rate

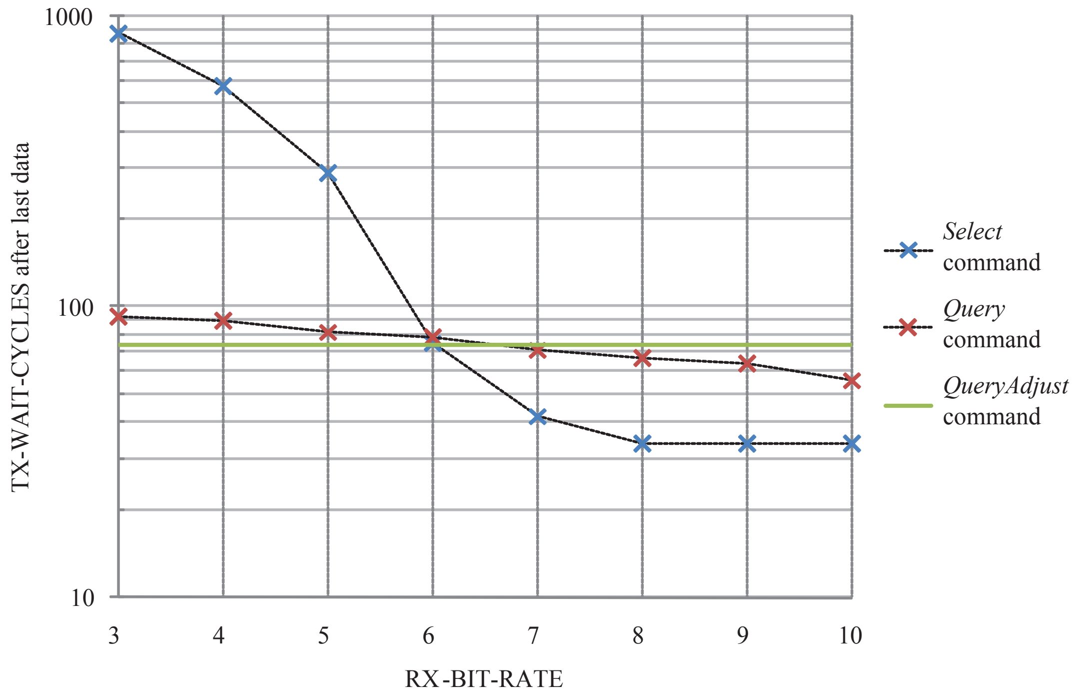

Figure 6.1: Required TX-WAIT-CYCLES depending on the RX-BIT-RATE for the *Select*, *Query* and *QueryAdjust* command. The *QueryAdjust* command is handled during the time T_1 in table 4.1, so it has a constant TX-WAIT-CYCLES requirement.

Table 6.7: Effective clock-rate for reception.

Tari[μs]	**Data-1 length**	**Clock-rate** [kHz]		
		Worst case	Nominal	Best case
6.5	1.5× data-0	1120	896.00	746.67
12.50	1.5× data-0	560	448.00	373.33
25.00	1.5× data-0	280	224.00	186.67
6.25	2× data-0	1120	746.67	560.00
12.50	2× data-0	560	373.33	280.00
25.00	2× data-0	280	186.67	140.00

Since the EPC Gen2 standard specifies flexible data rates for transmission and reception, the effective clock rate depends on the particular configuration. Table 6.7 shows the effective clock rates for receiving with a RX-BIT-RATE of 7 cycles per bit. The data-1 symbol length in Table 6.7 is given as a multiple of the data-0 symbol length. Because of the different data-0 and data-1 symbol lengths the effective clock rate depends also on the data itself. In the worst case only data-0 symbols are expected whereas in the best case only data-1 symbols are expected. In the average case an equal distribution of data-1 and data-0 symbols is assumed. The factor between the lowest and the highest effective clock rate is 4.8.

Table 6.8 shows the effective clock rates for transmitting with a TX-BIT-RATE of 5 cycles per bit. Because of the different encoding formats, the factor of 128 between the highest and the lowest effective clock rate is more significant than in table 6.7.

Table 6.8: Effective clock-rate for transmission.

BLF[kHz]	**Clock-rate[kHz]**			
Data encoding format	FM0	Miller		
		M=2	M=4	M=8
40	200.0	100.0	50.0	25.0
107	535.0	267.0	133.8	66.9
160	800.0	400.0	200.0	100.0
256	1280.0	640.0	320.0	160.0
320	1600.0	800.0	400.0	200.0
640	3200.0	1600.0	800.0	400.0

These factors between the lowest and the highest possible data rates have a direct influence on the power consumption. This allows working in wider ranges of low data rates and in return high data rates when working in closer ranges, where the electromagnetic field is stronger.

The HIGH-CLOCK is independent of the data-rate. But since it is switched on and off at certain points of the source code, it is only active for the maximum required TX-WAIT-CYCLES count, which is 74.

Chapter 7

Concluding Remarks

The goal of this thesis was to build a design based on a standard microcontroller with an additional front-end that handles RFID communication according to the EPC Gen2 standard. The microcontroller is based on the AVR instruction set of the Atmega128 with an 8-bit memoryinterface and an 8-bit datapath. The idea was to handle as much as possible on the microcontroller, to maximize the flexibility of the design.

When analyzing the communication protocol of the EPC Gen2 standard it seems that it is more suitable for a custom processor design, which is based on a state machine. Moreover the EPC Gen2 standard describes certain states for a tag, depending on multiple parameters and values. Most of the computation for the command handling can be parallelized in a state machine design. Thus handling the commands with software running on a standard microcontroller needs innately a higher clock rate than with a state machine. An elaborate software and also interface for the front-end are required to reduce this clock rate to a minimum.

This is achieved by implementing different clock rates for reception and transmission to avoid idle operations within certain parts of the source code. Therefore, the software for the command handling is also highly optimized in terms of required cycles for the critical commands during reception and transmission. The effect is that the required clock cycles are reduced to 7 cycles per received bit and 5 cycles per transmitted bit. Nevertheless, the applied software optimizations lead to a larger code size which is reflected in the large program memory (more than 50% of the chip area).

The microcontroller is clocked with exactly the amount of cycles it needs to process the data within the time constraints. This makes the effective clock rate dependent on the chosen data rates. With the lowest data rate during reception the effective clock is 186.67 kHz on average. With the lowest data rate during transmission the effective clock is 25kHz when Miller M=8 data encoding is used. Reducing the effective clock rates with lower data rates allows the tag to work in wider ranges with the interrogator, since it needs less power. On the other hand a reader is able to use higher data rates within close distances.

To verify that the design conforms to the EPC Gen2 protocol the whole design was implemented in real hardware using a demonstration tag with a Field Programmable Gate Array (FPGA). Therefore an EPC Gen2 reader was plugged in to a computer so that various communication scenarios could be tested.

The current design offers a fundament for many further implementations and extensions of the EPC Gen2 standard. Different aspects for security within an EPC Gen2 tag

can be easily elaborated on and evaluated by additional software. In a practical implementation based on this design a certain security feature or an update of the EPC Gen2 standard could be applied to an exiting tag by downloading new software.

The EPC Gen2 standard specifies a comparatively large command length for custom commands, so it is to be expected that for further source code extensions the critical and limiting commands for the effective clock rate of the microcontroller will remain the same. Nevertheless, one could consider swapping certain functionality into hardware. Therefore two approaches shall be mentioned. One approach would be an extension of the instruction set of the microcontroller. Another approach would be to preprocess the incoming data or to aftertreat the outgoing data inside the front-end, which, because of its modular design, should not require great modifications of the front-end or the interface to the microcontroller. However such extensions shall remain as a topic of further investigations and research.

Appendix A

Definitions

A.1 Acronyms

AES	Advanced Encryption Standard.
BLF	Backscatter-Link frequency.
CBC	Cipher Block Chaining.
CRC	Cyclic Redundancy Check.
DES	Data Encryption Standard.
DSB-ASK	Double-Sideband Amplitude Shift Keying.
ECB	Electronic Code Book.
EPC	Electronic Product Code.
FIFO	”First in, First out”-Queue.
FSR	Feedback Shift Register.
HDL	Hardware Description Language.
HF	High Frequency.
ISO	International Organization of Standardization.
ITF	Interrogator-Talks-First.
LF	Low-Frequency.
LFSR	Linear-Feedback-Shift-Register.
LSB	Least Significant Bit.
NIST	National Institute of Standards and Technology.
OFB	Output-Feedback.

PIE	Pulse Interval Encoding.
PR-ASK	Phase-Reversal Amplitude Shift Keying.
PRNG	Pseudo Random Number Generator.
RFID	Radio-Frequency Identification.
RN16	16-bit Random or Pseudo-Random Number.
RX-BIT-RATE	Number of Cycles per Received Bit.
SSB-ASK	Single-Sideband Amplitude Shift Keying.
TX-BIT-RATE	Number of Cycles per Transmitted Bit.
TX-WAIT-CYCLES	Constant Amount of Clock Cycles between Reception and Transmission.
UHF	Ultra High Frequency.

Appendix B

Interface Register

B.1 Status 1 Register

Bit number	7	6	5	4
Name	tx_mod_stop	fifo_not_empty	rx_eof	cmd_detected
Bit number	3	2	1	0
Name	query_adj_cmd	query_cmd	ack_cmd	query_rep_cmd

Table B.1: Flags in the status_1 register

The flags in the status_1 register, as shown in table B.1, are set and cleared by the frontend to invoke the right handling by the microcontroller. Some of them are cleared by the microcontroller to ensure a correct control flow of the frontend. Their function in particular is:

- **tx_mod_stop**
  This flag has two functions. When the tx_enable flag is set by the microcontroller, the frontend sets the tx_mod_stop flag, when the last bit is transmitted and the modulation is finished. When the crc_enable flag is set by the microcontroller, the frontend sets the tx_mod_stop flag, when the CRC-16 is calculated over all the data pushed in the FIFO. Furthermore it tells the microcontroller that the calculated CRC-16 is already pushed into the FIFO.

- **fifo_not_empty**
  This flag is set if there is minimum one byte in the FIFO and cleared if it is empty.

- **rx_eof**
  It is set by the frontend, when the last received bit is pushed into the FIFO. The frontend clears the rx_eof flag either if the time T_1 in table 4.1 is over and tx_enable is set, or if rx_enable is cleared and tx_enable is not set.

- **cmd_detected**
  If receiving is enabled, this flag is set if a command is detected by the frontend. Either it is one of the five special commands and hence an additional flag for the specific command is set, or it is one of the other miscellaneous commands and no additional flag is set (except for the *NAK* command). Thus the flag is set at latest when the fourth bit signal is received.

- **query_adj_cmd**
  This flag is set together with the `cmd_detected` flag, if the detected command was the *QueryAdjust* command

- **query_cmd**
  This flag is set together with the `cmd_detected` flag, if the detected command was the *Query* command

- **ack_cmd**
  This flag is set together with the `cmd_detected` flag, if the detected command was the *Ack* command

- **query_rep_cmd**
  This flag is set together with the `cmd_detected` flag, if the detected command was the *QueryRep* command

B.2 Status 2 Register

Bit number	7	6	5	4
Name	RFU	RFU	`last_data_invalid`	`fifo_full`
Bit number	3	2	1	0
Name	`nak_cmd`	`select_cmd`	`crc16_valid`	`crc5_valid`

Table B.2: Flags in the status_2 register

The flags in the status_2 register, as shown in table B.2, are set and cleared only by the frontend. Their function in particular is:

- **last_data_invalid**
  If EOF is detected during reception the front-end checks if it received the expected amount of data. This amount is signalized via the `push_after_size` bits in the control_2 register by the microcontroller. If the amount was incorrect, the `last_data_invalid` flag is set. It is cleared again, during the polling of a new incoming command (whit the rising edge of the `rx_enable` flag)

- **fifo_full**
  This flag is set if theFIFO is full and no other byte can be pushed into it. It is cleared if at least one push is possible.

- **nak_cmd**
  This flag is **not** set together with the `cmd_detected` flag. Another four data symbols are required to detect the *NAK* command and thus to set this flag.

- **select_cmd**
  This flag is set together with the `cmd_detected` flag, if the detected command was the *Select* command

- **crc16_valid**
  Every received bit is shifted into a shift register, that calculates the CRC-16. This

register is preset to the value $FFFF_h$ when the `rx_enable` flag is set. If within a shift operation the value in the register is $1D0F_h$, the last 16 bits are a valid CRC-16 value over the bits received before the last 16 bits and the `crc16_valid` flag is set. This can also occur randomly in the middle of a data stream, so this flag only makes sense, if the `rx_eof` was set.

- **`crc5_valid`**
  Every received bit is shifted into a shift register, that calculates the CRC-5. This register is preset to the value 01001_2 when the `rx_enable` flag is set. If within a shift operation the value in the register is 00000_2, the last 5 bits are a valid CRC-5 value over the bits received before the last 5 bits and the `crc5_valid` flag is set. This can also occur randomly in the middle of a data stream, so this flag only makes sense, if the `rx_eof` was set.

B.3 Control 1 Register

Bit number	7	6	5	4
Name	`write_mem`	`stored_crc_enable`	`tx_header_value`	`add_tx_header`
Bit number	3	2	1	0
Name	`add_tx_crc`	`tx_conf_store`	`rx_enable`	`tx_enable`

Table B.3: Flags in the control_1 register

The flags in the control_1 register, as shown in table B.3, are set and cleared by the microcontroller to enable and adjust different functions of the front-end. Their function in particular is:

- **`write_mem`**
  This flag is set by the microcontroller, if the tag has to write to the memory, as a response to a command. This may be provoked by a *Write*,*Lock* or *Kill* command. If the flag is set, the frontend may exceed the T_1 value in table 4.1 according to [11]. It is also used to set the intern `TRext` value of the frontend (see [11]).

- **`stored_crc_enable`**
  This flag is used to tell the front-end that it shall calculate the CRC-16 over the data being pushed into the FIFO. The data will not be transmitted.

- **`tx_header_value`**
  If this flag is set, the preceding header bit is 1_2 for the transmitted data. Otherwise the header bit is 0_2. This flag is only valid if the `add_tx_header` bit is set.

- **`add_tx_header`**
  This flag shall be set with the `tx_enable` flag, if the data to be transmitted has to be preceded by a header bit. The value of the header bit is given by the `tx_header_value` flag.

- **`add_tx_crc`**
  This flag is set, if a CRC-16 has to be added to the transmitted data. The frontend

starts to transmit the value in the CRC-16 shift register, when the FIFO is empty and the `tx_eof` flag is set.

- **`tx_conf_store`**
  This flag is used to save the configurations values of the frontend. It is set by the microcontroller after detecting a valid *Query* command for just one cycle. If the frontend detects a possible *Query* command, it extracts the DR,M and TRext parameter and stores these temporarily, while receiving the rest of the command. When the `tx_conf_store` is set, it overwrites the old configuration values with the temporary ones.

- **`rx_enable`**
  This flag is set to enable receiving by the frontend. Otherwise no data will be pushed into the FIFO. It has to be cleared before the time T_1 in table 4.1 is over.

- **`tx_enable`**
  This flag is set to enable transmission by the frontend. It has to be set before the time T_1 in table 4.1 is over. When the flag is set, the front-end starts transmitting the preamble. When finished with the preamble it starts transmitting the data pushed into the FIFO.

B.4 Control 2 Register

Bit number	7	6	5	4
Name	`hi_clk`	`push_after_size`		
Bit number	3	2	1	0
Name	`tx_eof`	`last_byte_size`		

Table B.4: Flags in the control_2 register

The flags in the control_2 register, as shown in table B.4, are set and cleared by the microcontroller to enable and adjust different function of the front-end. Their function in particular is:

- **`hi_clk`**
  If this flag is set, the microcontroller requests the highest clock from the front-end.

- **`push_after_size`**
  These three bits represent the number of the received bits, after which the front-end should push currently received data into the FIFO. If the number is 000_2, then the full byte is waited for until the data is pushed into the FIFO.

- **`tx_eof`**
  If the `tx_enable` flag is set, this flag is set by the microcontroller to tell the front-end that no more data will be transmitted, than the one pushed into the FIFO. If the `crc_enable` flag is set, this flag is tells the front-end, that the data for the CRC-16 calculation, has been completely pushed into the FIFO.

- **`last_byte_size`**
  These three bits represent the size of the valid bits of the last byte in the FIFO, beginning with the Least Significant Bit (LSB).

- $000_2 \rightarrow 1$ bit is valid
- $001_2 \rightarrow 2$ bits are valid
- $010_2 \rightarrow 3$ bits are valid
- $011_2 \rightarrow 4$ bits are valid
- $100_2 \rightarrow 5$ bits are valid
- $101_2 \rightarrow 6$ bits are valid
- $101_2 \rightarrow 7$ bits are valid
- $111_2 \rightarrow 8$ bits are valid

These bits are used in both directions. When the microcontroller pushes the last data into the FIFO and sets the `tx_eof` flag, it also sets these bits accordingly. On the other side, when the front-end pushed the last received data into the FIFO and sets the `rx_eof` flag, it also sets these bits accordingly.

Appendix C

Algorithms

C.1 Extended Euclidean algorithm

The extended Euclidean algorithm for two non-negative integers a and b with $a \geq b$ delivers $d = gcd(a,b)$ and x,y such that $a \times x + b \times y = d$ and can be executed as follows:

1. If $b = 0$ then set $d \leftarrow a$, $x \leftarrow 1$, $y \leftarrow 0$; finish
2. Set $x_1 \leftarrow 0$, $x_2 \leftarrow 1$, $y_1 \leftarrow 1$, $y_2 \leftarrow 0$.
3. While $b > 0$ do:
   (a) $q \leftarrow \lfloor a/b \rfloor$, $r \leftarrow a - q \times b$, $x \leftarrow x_2 - q \times x_1$, $y \leftarrow y_2 - q \times y_1$.
   (b) $a \leftarrow b$, $b \leftarrow r$, $x_2 \leftarrow x_1$, $x_1 \leftarrow x$, $y_2 \leftarrow y_1$, $y_1 \leftarrow y$.
4. Set $d \leftarrow a$, $x \leftarrow x_2$, $y \leftarrow y_2$; finish

Bibliography

[1] S. A. Weis, “Rfid (radio frequency identification): Principles and applications,” 2007. [Online]. Available: http://www.eecs.harvard.edu/cs199r/readings/rfid-article.pdf (Cited on pages 1, 5 and 6.)

[2] M. Roberti, “Epc reduces out-of-stocks at wal-mart,” 2005. [Online]. Available: http://www.rfidjournal.com/articles/view?1927/2 (Cited on pages 1 and 6.)

[3] W. Dong-Liang *et al.*, “A brief survey on current rfid applications,” vol. 4, 2009, pp. 2330–2335. (Cited on pages 1, 5 and 6.)

[4] I. Lacmanovic, B. Radulovic, and D. Lacmanovic, “Contactless payment systems based on rfid technology,” in *MIPRO, 2010 Proceedings of the 33rd International Convention*, May 2010, pp. 1114–1119. (Cited on pages 1 and 7.)

[5] A. Jeffries, “Internet of cows: technology could help track disease, but ranchers are resistant,” May 2013. [Online]. Available: http://www.theverge.com/2013/5/10/4316658/internet-of-cows-technology-offers-ways-to-track-livestock-but (Cited on pages 1 and 6.)

[6] A. Juels, “Rfid security and privacy: A research survey,” *JOURNAL OF SELECTED AREAS IN COMMUNICATION (J-SAC)*, vol. 24, no. 2, pp. 381–395, 2006. (Cited on pages 1, 7 and 9.)

[7] R. Pateriya and S. Sharma, “The evolution of rfid security and privacy: A research survey,” in *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, June 2011, pp. 115–119. (Cited on pages 1 and 9.)

[8] E. Bogari, P. Zavarsky, D. Lindskog, and R. Ruhl, “An analysis of security weaknesses in the evolution of rfid enabled passport,” 2012, pp. 158–166. (Cited on pages 1 and 8.)

[9] N. Bagheri, M. Safkhani, P. Peris-Lopez, and J. E. Tapiador, “Comments on ”security improvement of an rfid security protocol of iso/iec wd 29167-6”.” *IEEE Communications Letters*, vol. 17, no. 4, pp. 805–807, 2013. (Cited on pages 1 and 10.)

[10] B. Song, J. Y. Hwang, and K.-A. Shim, “Security improvement of an rfid security protocol of iso/iec wd 29167-6.” *IEEE Communications Letters*, vol. 15, no. 12, pp. 1375–1377, 2011. (Cited on pages 1 and 10.)

[11] EPCglobal Inc., “Uhf class 1 gen 2 standard v. 1.2.0,” Oct. 2008. [Online]. Available: http://www.gs1.at/images/stories/Produkte/GS1_EPCglobal/EPC_Standards/UHF_C1_Gen2/GS1_EPC_uhfc1g2_1_2_0-standard-20080511.pdf (Cited on pages 2, 24, 25, 26, 27, 28, 30, 35 and 64.)

[12] J. Landt, "The history of rfid," *IEEE Potentials*, vol. 24, no. 4, pp. 8–11, 2005. (Cited on page 5.)

[13] B. Violino, "The history of rfid technology," *RFID Journal*, 2005. [Online]. Available: http://www.rfidjournal.com/articles/view?1338/2 (Cited on pages 5 and 6.)

[14] ISO/IEC 18000-3, "Information technology – radio frequency identification for item management – part 3: Parameters for air interface communications at 13,56 mhz," 2013. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=53424 (Cited on page 5.)

[15] ISO/IEC 18000-6, "Information technology – radio frequency identification for item management – part 6: Parameters for air interface communications at 860 mhz to 960 mhz general," 2013. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=59644 (Cited on pages 5 and 10.)

[16] Z. Yuan and D. Huang, "A novel rfid-based shipping containers location and identification solution in multimodal transport," in *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*, May 2008, pp. 000 267–000 272. (Cited on page 6.)

[17] CASPIAN, "Position statement on the use of rfid on consumer products," 2003. [Online]. Available: https://w2.eff.org/Privacy/Surveillance/RFID/RFID_Position_Statement.pdf (Cited on page 8.)

[18] S. C. Bono, M. Green, A. Stubblefield, A. Juels, A. D. Rubin, and M. Szydlo, "Security analysis of a cryptographically-enabled rfid device," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251399 (Cited on page 9.)

[19] A. Man, E. Zhang, V. Lau, and C. Tsui, "Low power vlsi design for a rfid passive tag baseband system enhanced with an aes cryptography engine," 2007, pp. 1–6. (Cited on pages 10 and 33.)

[20] A. Ricci, M. Grisanti, I. De Munari, and B. Ciampolini, "Design of a 2 µw rfid baseband processor featuring an aes cryptography primitive," 2008, pp. 376–379. (Cited on pages 10 and 33.)

[21] ISO/IEC WD 29167-6, "Information technology – automatic identification and data capture techniques – part 6: Air interface for security services and file management for rfid at 860-960mhz," 2011. [Online]. Available: http://www.iso.org (Cited on page 10.)

[22] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996. (Cited on pages 11, 12, 16 and 17.)

[23] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley, 2008. (Cited on page 11.)

[24] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*, 2nd ed. New York: Wiley, 1996. (Cited on pages 11, 15 and 22.)

[25] J. Yu and P. Brune, "No security by obscurity - why two factor authentication should be based on an open design," in *Security and Cryptography (SECRYPT), 2011 Proceedings of the International Conference on*, July 2011, pp. 418–421. (Cited on page 12.)

[26] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978. (Cited on page 17.)

[27] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," NIST Web page, March 1999. [Online]. Available: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf (Cited on pages 18, 20 and 21.)

[28] R. M. Stallman and G. D. Community, *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009. (Cited on page 35.)

[29] J. Melia-Segui, J. Garcia-Alfaro, and J. Herrera-Joancomarti, "Multiple-polynomial lfsr based pseudorandom number generator for epc gen2 rfid tags," in *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, Nov 2011, pp. 3820–3825. (Cited on page 40.)

[30] T. A. Roth *et al.*, "Simulavr - an avr simulator," Feb. 2012. [Online]. Available: http://www.nongnu.org/simulavr/ (Cited on page 49.)

[31] "Gnu general public license," Free Software Foundation, 2007. [Online]. Available: http://www.gnu.org/licenses/gpl.html (Cited on page 49.)

[32] ISO/IEC WD 29167-10, "Information technology – automatic identification and data capture techniques – part 10: Air interface for security services crypto suite aes128," 2011. [Online]. Available: http://www.iso.org (Cited on page 50.)

[33] C. Meiß, "Rfid - logistics and supply chain management," *RFID Systems and Technologies (RFID SysTech)*, pp. 1–7, 2007. (Not cited.)

[34] T. A. Roth *et al.*, "Simulavr - an avr simulation framework," Feb. 2012. [Online]. Available: http://download.savannah.gnu.org/releases/simulavr/manual-1.0.pdf (Not cited.)