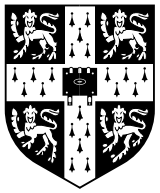
The Smart Card Detective: a hand-held EMV interceptor

Omar S. Choudary



University of Cambridge
Computer Laboratory
Darwin College

June 2010

This dissertation is submitted for the degree of
Master of Philosophy in Advanced Computer Science

Declaration

I Omar Salim Choudary of Darwin College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

The word count, including footnotes, bibliography and appendices is 14 978.

Signed:

Date:

The Smart Card Detective: a hand-held EMV interceptor

Omar Choudary

Abstract

Several vulnerabilities have been found in the EMV system (also known as Chip and PIN). Saar Drimer and Steven Murdoch have successfully implemented a relay attack against EMV using a fake terminal. Recently the same authors have found a method to successfully complete PIN transactions without actually entering the correct PIN. The press has published this vulnerability but they reported such scenario as being hard to execute in practice because it requires specialized and complex hardware.

As proposed by Ross Anderson and Mike Bond in 2006, I decided to create a miniature man-in-the-middle device to defend smartcard users against relay attacks.

As a result of my MPhil project work I created a hand-held device, called **Smart Card Defender** (SCD), which intercepts the communication between smartcard and terminal. The device has been built using a low cost ATMEL AT90USB1287 microcontroller and other readily available electronic components. The total cost of the SCD has been around £100, but an industrial version could be produced for less than £20.

I implemented several applications using the SCD, including the defense against the relay attack as well as the recently discovered vulnerability to complete a transaction without using the correct PIN.

All the applications have been successfully tested on CAP readers and live terminals. Even more, I have performed real tests using the SCD at several shops in town.

From the experiments using the SCD, I have noticed some particularities of the CAP protocol compared to the EMV standard. I have also discovered that the smartcard does not follow the physical transport protocol exactly. Such findings are presented in detail, along with a discussion of the results.

Acknowledgments

I thank my supervisor, Markus Kuhn, for extensive guidance and valuable advice on rigorous design and research.

I am grateful to Mike Bond and Steven Murdoch for many useful discussions on EMV.

Saar Drimer and Sergei Skorobogatov have been extremely helpful with hardware advice and even hands-on support.

Thanks also to Frank Stajano for suggesting this very exciting project and to Ross Anderson for the trust and advice.

Last I thank my wife Daniela, for all the moral support during hard times.

Thank you all for an extraordinary and challenging experience.

Contents

1 Introduction **7**

2 Background **9**

2.1 ISO 7816 11

2.1.1 ISO 7816-2: dimensions and locations of the contacts 11

2.1.2 ISO 7816-3: electronic signals and transmission protocols 11

2.2 EMV 14

2.2.1 Transmission of commands and responses 14

2.2.2 Transaction flow 16

3 Related work **18**

4 SCD overview **20**

4.1 Hand-held device 20

4.2 Applications 22

5 SCD implementation **23**

5.1 Requirements and constraints 23

5.2 Hardware 24

5.2.1 ATMEL AT90USB1287 AVR microcontroller 25

5.2.2 Terminal and smartcard interface 26

5.2.3 Power sources 27

5.2.4 Peripherals 28

5.2.5 Prototype 30

5.2.6 PCB 32

5.3 Software 35
5.3.1 Architecture 36
5.3.2 Initialization sequence 38
5.3.3 Interrupts and power down modes 39
5.3.4 Memory 40
5.3.5 Operation 40
5.4 Terminal emulator 42

6 Evaluation **44**
6.1 Basic functionality 44
6.2 Power consumption 45
6.3 Functionality tests 47

7 Conclusion **51**

References **53**

Appendix **55**

A Source code for byte transmission **55**

Chapter 1

Introduction

Many banks across Europe have introduced a new payment system, EMV, also known as Chip and PIN in the UK. EMV is a complex standard that defines the protocol used between a point of sale terminal and a smartcard. I provide a general understanding of EMV and details of the parts related to my project in the next chapter.

In a normal payment scenario (e.g. purchasing food at a supermarket) the terminal is owned by the supermarket and the smartcard is owned by the **issuer** bank. Thus the card **user** has no control over the transaction entities. In this scenario it is possible for someone to tamper with the terminal such that the amount shown on the display is higher than the amount requested to the card. The user will confidently enter the PIN and authorize the transaction.

Financial fraud in the UK has not decreased over the last years, even after Chip and PIN has been introduced. According to APACS [3] the overall level of frauds has remained relatively the same, even if the level of particular types of fraud has changed. Murdoch et al. [22] suggest that EMV has simply moved fraud, but not eliminated it. Therefore the discovery of vulnerabilities and the development of solutions against financial fraud remains an important need.

It was the main goal of this project to create a man-in-the-middle device, called *Smart Card Detective* (SCD), that would be able to prevent the attack described above. Such a device would have to intercept the communication between a card and a terminal, provide the user with the ability to observe the amount requested by the terminal, and then continue or reject the transaction based on the user decision.

Another goal of this project was to build a device that is small enough to hold it in a hand like you hold a mobile phone, and cheap enough for many users to actually afford. The motivation for this was to prove that a miniature device, able to perform man-in-the-middle EMV operations such as protecting users against fraud can be built.

After first creating a prototype board to prove the correctness of my design, I managed

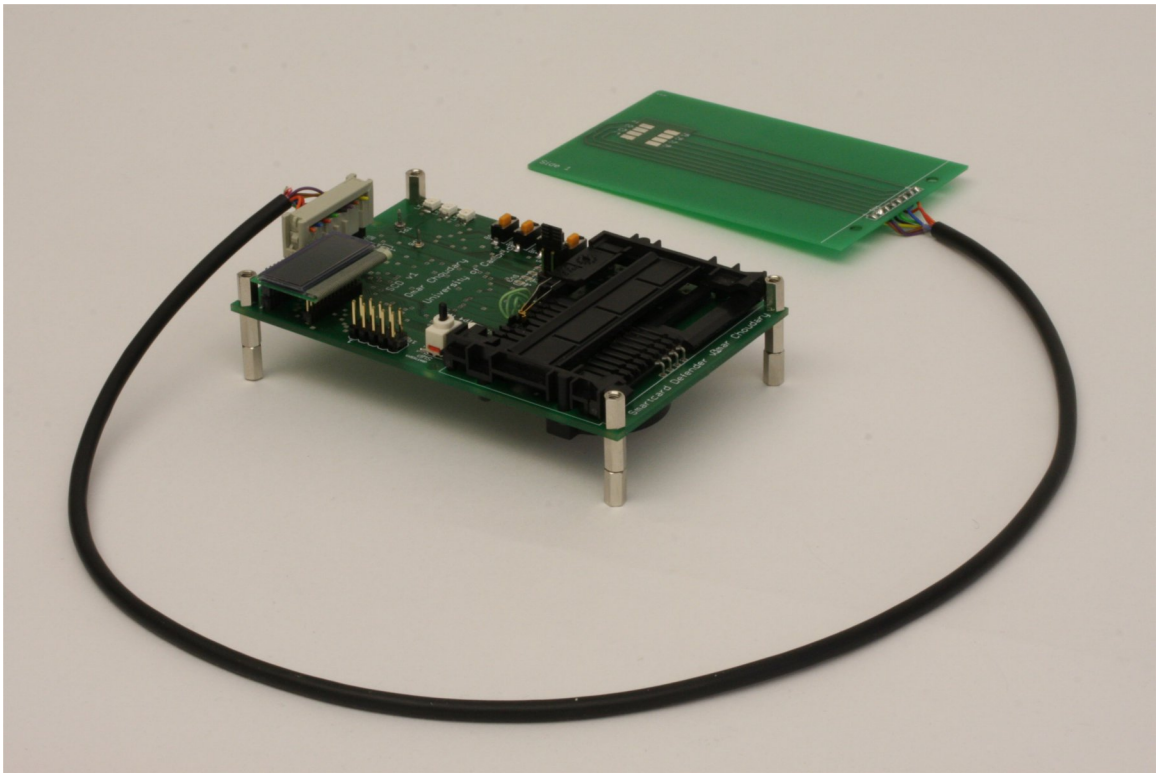


Figure 1.1: *The Smart Card Detective*

to create a hand-held device that provides a trusted display for a transaction and allows users to defend against relay attacks. In figure 1.1 you can see an image of the final device. An overview of the SCD is presented in chapter 4 while the details of the implementation are presented in chapter 5.

A recently discovered vulnerability allows the use of any card (possibly stolen) without knowing its PIN. The French press has published news about the attack but they reported the scenario as being hard to execute in practice because it required specialized and complex hardware.

I have managed to implement this attack on the SCD quite easily as I already had a robust framework in place which made the implementation straightforward. This proves that a small and cheap device is able to tamper with a complex system such as EMV, but also suggests that technically competent criminals may have already developed their own devices.

The results of experiments using multiple cards and readers as well as some interesting findings are presented in chapter 6.

Chapter 2

Background

Chip and PIN is the popular UK name for the payment system used in many countries across Europe. In this system the banks (**issuers**) provide their clients (**users**) with a smartcard that can be used to withdraw money from ATMs, make payment transactions and even authenticate online transactions. In the remainder of this document I will discuss only the last two scenarios.

A normal payment transaction requires the user to insert the smartcard (figure 2.1(a)) into a point of sale terminal (figure 2.1(b)) and enter a PIN number (usually 4 digits, but possibly longer) to authorize the transaction. The PIN number is provided by the bank but can usually be changed by the user at any time using an ATM.

(a) (b) (c)

Figure 2.1: *Entities involved in the Chip and PIN payment system: smartcard (a), terminal (b) and CAP reader (c)*

The formal name of the protocol defining the rules for the communication between the smartcard and the terminal is **EMV**. This stands for Europay, MasterCard and VISA,

the organizations involved in the original design of the protocol.

EMV relies on the ISO-7816 standard [20] which defines the general characteristics of Integrated Circuit(s) with Contacts (ICC - generally referred to as smartcard). However EMV is a complex protocol, with a base specification that spans 4 books [12–15]. In addition each country and bank has developed particular protocols on top of the reference specification. This has led to some vulnerabilities in the overall solution, as shown by the attacks presented in the next chapter.

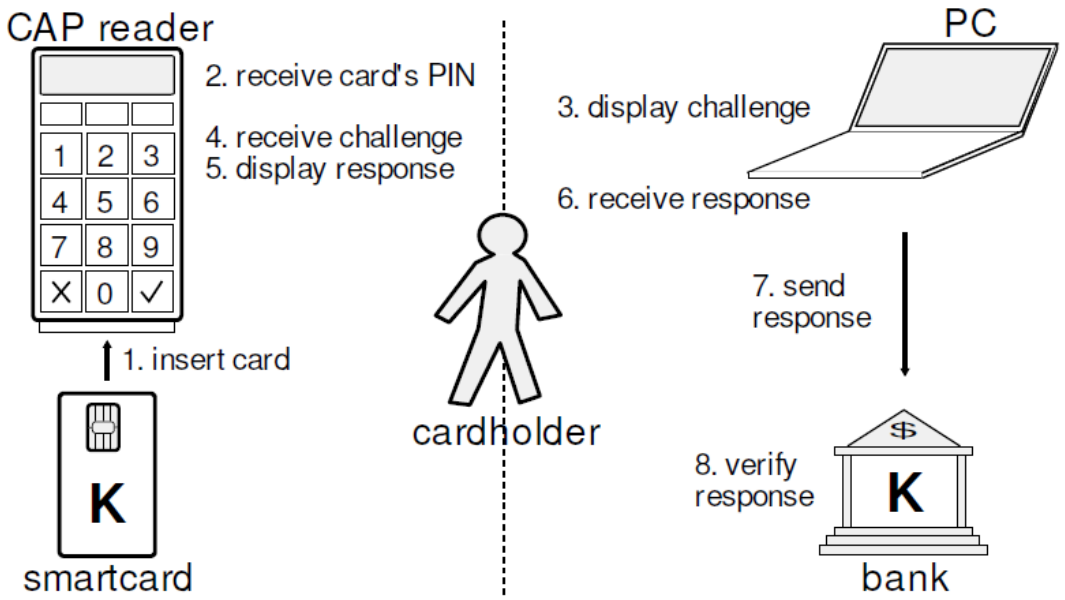
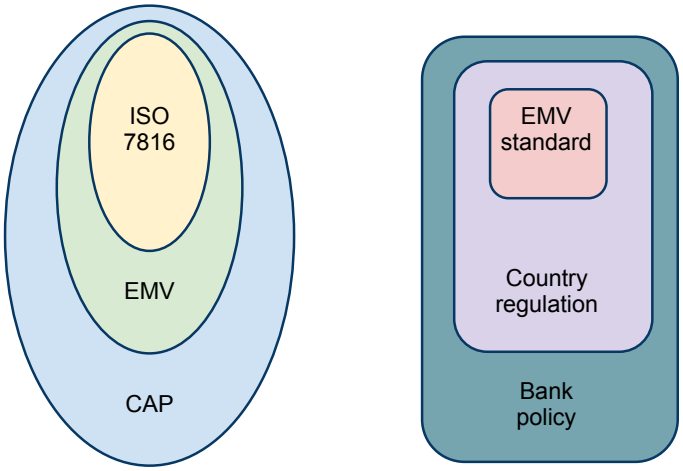


Figure 2.2: *Online authentication using CAP. Image from Steven Murdoch, used in Optimized to Fail: Card Readers for Online Banking*

Some banks in the UK have implemented the Chip Authentication Program (CAP) [21], to authenticate online transactions. A typical scenario requires the user to access a bank's web page, insert the smartcard into a CAP reader (see figure 2.1(c)), enter the PIN, and get from the CAP reader a code that must be typed on the web page to complete the authentication. An illustration of a general approach is provided in figure 2.2.

CAP uses EMV but adds its own functionality to the standard protocol. Together with a proprietary implementation by each bank this adds to the overall complexity and the risk of vulnerabilities.

Figure 2.3(a) illustrates the hierarchy of protocols used in the Chip and PIN system and figure 2.3(b) shows the different layers of specifications used with EMV.



(a) (b)

Figure 2.3: *Main protocols used in Chip and PIN (a); EMV international hierarchy (b)*

2.1 ISO 7816

The ISO 7816 standard is composed of ten parts that define physical characteristics, electronic signals and prototocols. For the purpose of understanding the EMV protocol and the work presented in this document only parts 2 and 3 are of interest and are described below.

2.1.1 ISO 7816-2: dimensions and locations of the contacts

A smartcard has eight contacts, labeled C1 through C8. This part of the standard defines the size and position of these contacts relative to the card.

The normal size of a card has a width of 85.6 mm and a height of 54 mm.

The position of the contacts is presented in figure 2.4. The meaning of each contact is shown in table 2.1.

2.1.2 ISO 7816-3: electronic signals and transmission protocols

The third part of the ISO 7816 standard defines the voltage thresholds for each contact as well as the protocols used to transmit data between the terminal and card.

The reference voltage is the supply voltage (V_{CC}), given by the terminal. Under normal conditions V_{CC} should be between 4.75 and 5.25 V, and the maximum current I_{CC} should be limited to 200 mA.

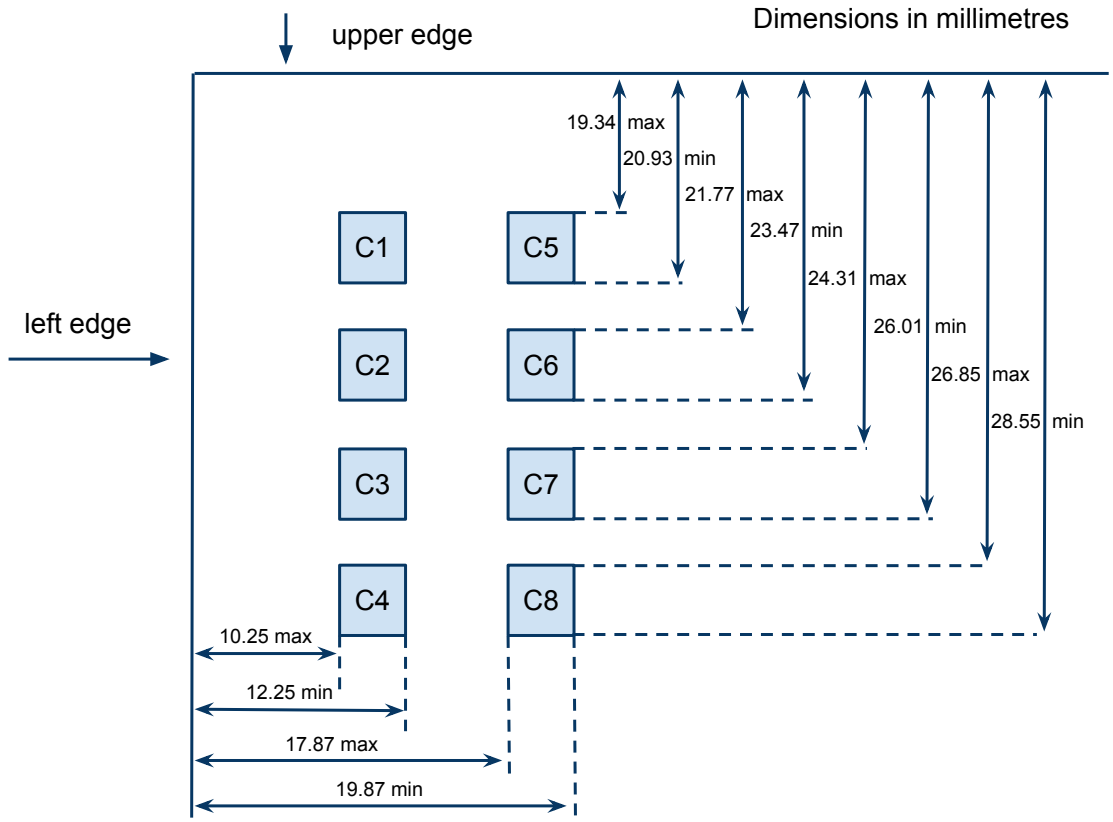


Figure 2.4: *Location of the contacts as defined in ISO 7816-2*

Table 2.1: Assignment of the contacts as defined in ISO 7816-2

Contact	**Assignment**	**Contact**	**Assignment**
C1	VCC (Supply voltage)	C5	GND (Ground)
C2	RST (Reset signal)	C6	VPP (Variable supply voltage)
C3	CLK (Clock signal)	C7	I/O (Data input/output)
C4	Reserved for future use	C8	Reserved for future use

The communication between the card and the terminal is asynchronous, meaning that only one of them can use the I/O line to transmit data at a given time, but not both. The I/O line can be in two states: high (state Z - voltage above 2 V) or low (state A - voltage below 0.8 V). When there is no communication, the I/O line should be held in state Z. If one of the sides wants to transmit data, it will put the I/O line in state A.

In order to initiate the communication, the terminal must issue a reset to the card. This procedure is as follows: first the voltage V_{CC} is enabled (and optionally V_{PP}), the reset line is set to low, and then clock is applied. Within 200 clocks the I/O line is

set to state Z. After 40000 clock cycles the reset line is set to high and the card should reply with a sequence of bytes known as *Answer to Reset* (ATR). A reverse procedure, called *deactivation of contacts* is used when the card is removed from the terminal or a transaction is ended.

The characters returned by the ATR provide information about: the format of each character, the transport protocol, the elementary time unit (ETU), the minimum and maximum delay between characters, and optionally a check sum.

The ETU specifies the bit duration in terms of terminal clock cycles. The default value is 372 clocks, but this can be changed by setting a different value in character $TA1$ of the ATR. The sender should ensure a precise bit duration and the receiver should read the bit value about mid-time of the ETU.

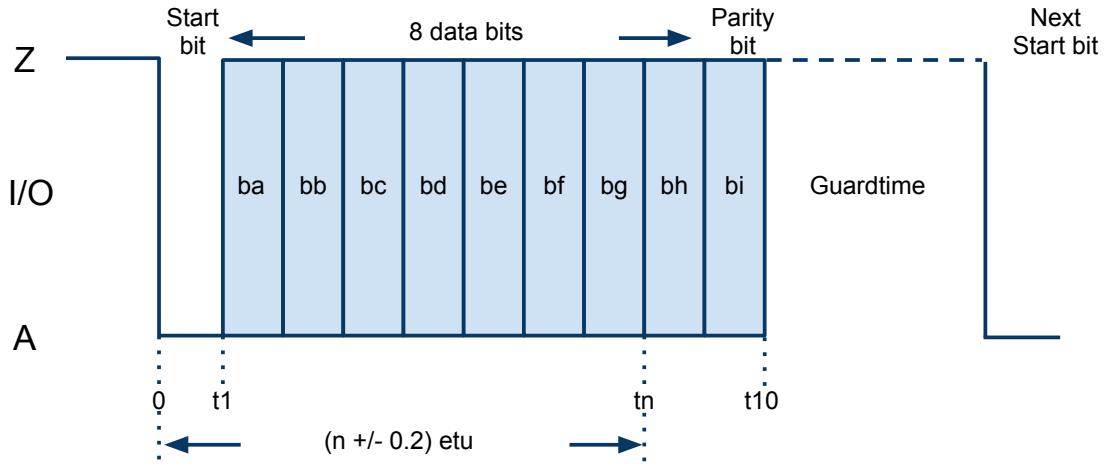


Figure 2.5: *ISO 7816-3 character frame*

Each character is transmitted as a series of 10 bits like in figure 2.5. The first bit (state A) is called the start bit and is used to signal the start of a byte transmission. The next 8 bits (ba through bh) represent the contents of the data byte, and should be interpreted according to the convention in use as described below. The last bit (bi) is called the parity bit and is used to check that there is an even number of ONES (state A or B depending on convention) in the 9 bits of data (ba through bi). If this test fails a parity error has occurred. In such case the bad character will be retransmitted if the transport protocol in use is $T = 0$. The protocol $T = 1$ has a different mechanism to detect errors by means of block check sums.

The encoding of bytes can use either a direct or inverse convention, as specified in the byte **TS** of the ATR. For direct convention, TS is **AZZAZZZAAZ**, a logic ONE is represented by state Z and the most significant bit (msb) is **bh**. For inverse convention TS is **AZZAAAAAAZ**, a logic ONE is represented by state A and the msb is **ba**.

The transport protocol ($T = 0$ or $T = 1$) is determined by the bytes $T0$ and TD_1 of the ATR, and is used to exchange commands and responses. For the remaining of this document I will refer only to protocol $T = 0$ as it is the most commonly used.

The commands are composed of a command header and optional data. The header is represented by 5 bytes (CLA, INS, P1, P2 and P3), that uniquely define the command and the length of command data or expected data. Upon reception of a command the card should return a response under the control of procedure bytes (e.g. wait more time, send command again, send another command or error). If no errors occur the response should contain two status bytes ($SW1 = 0x90$ and $SW2 = 0x00$ if everything is fine) and optional data as required by the command. The next section describes in more detail the use of commands in the Chip and PIN system.

2.2 EMV

The EMV specification (version 4.2 at the time of writing this document) uses and extends parts of the ISO 7816 standard. The is to keep the compatibility with the ISO standard as much as possible while providing the necessary functionality.

The electrical characteristics are mostly the same as those specified in ISO 7816-2. The major difference is the specification of different power classes (class A with V_{CC} at 5V and I_{CC} min 55 mA, class B with V_{CC} at 3V and I_{CC} min 55 mA, and class C with V_{CC} at 1.8V I_{CC} min 35 mA) that should be supported by new terminals and cards. The purpose is to introduce terminals that support only class B from January 2014 in order to reduce power consumption.

Initialization of communication and transmission of characters is done according to the ISO 7816-3 standard, as explained in the previous section.

2.2.1 Transmission of commands and responses

The communication between the terminal and the card is done by transferring commands from terminal to card and responses from card to terminal. Some commands may have command data, and a response may have associated data depending on the command.

The byte sequence composed of a command header and the optional data is called Command Application Protocol Data Unit (C-APDU), and the sequence composed of the response bytes plus associated data is called Response Application Protocol Data Unit (R-APDU).

The commands used by EMV are split in four cases depending on the existence of data in the command and response. This is illustrated in table 2.2.

Table 2.2: Command cases in EMV

Case	Command Data	Response Data
1	Absent	Absent
2	Absent	Present
3	Present	Absent
4	Present	Present

Table 2.3: Examples of data transmission. Information extracted from EMV version 4.2 Book 1, Annex A sections A5 and A6

Case 2 command		**Case 4 command**	
terminal	**card**	**terminal**	**card**
[CLA INS P1 P2 00] =>		[CLA INS P1 P2 Lc] =>	
	<= 6C Licc		<= [INS]
[CLA INS P1 P2 Licc] =>		[Data(Lc)] =>	
	<= 61 xx		<= 61 xx
[00 C0 00 00 yy] =>		[00 C0 00 00 xx] =>	
	<= C0 [Data(yy)] 61 zz		<= C0 [Data(xx)] 61 yy
[00 C0 00 00 zz] =>		[00 C0 00 00 yy] =>	
	<= C0 [Data(zz)] 90 00		<= C0 [Data(yy)] 90 00
A R-APDU of **[Data(yy+zz)] 90 00** is returned from card to terminal		A R-APDU of **[Data(xx+yy)] 90 00** is returned from card to terminal	

The byte P3 in the command header represents the length of the command data or the length of the expected data depending on the command case.

For commands with data (cases 2 and 4), the command header must be sent, then the card must reply with a **procedure byte** (equal to INS indicating that all data may be sent or $\overline{INS}$ indicating that only the next byte should be sent) and only then the terminal should send the command data.

For cases 3 and 4 (response data expected), after receiving the command header and optionally the data (for case 4), the card will send the expected data under the control of procedure bytes as described above. Other possible procedure bytes are: $0x60$ requesting the terminal for additional work time, $0x61$ followed by $0xXX$ meaning that the terminal should issue a **GET RESPONSE** command ($CLA = 0x00$, $INS = 0xC0$) with $P3 = 0xXX$, and $0x6C$ followed by $0xXX$ meaning that the terminal should resend the previous command with $P3 = 0xXX$.

When a response (with or without data) is returned, the card will transmit two **status bytes** SW1 and SW2. The common value for success is $SW1 = 0x90$ $SW2 = 0x00$, but there are other possible values indicating a warning or error condition.

Two examples of command-response transmissions for case 2 and case 4 commands are shown in table 2.3. Next I describe a typical transaction flow and the commands involved.

2.2.2 Transaction flow

A transaction starts by selecting the desired application, which should be supported by both the card and the terminal. The command used for this purpose is **SELECT**, identified by $CLA = 0x00$ and $INS = 0xA4$. There are two ways to select an application: select by name using **1PAY.SYS.DDF01**, or select by Application Identifier (AID). The former requires a further **READ RECORD** command ($CLA = 0x00$, $INS = 0xB2$) to select the applications under the directory **1PAY.SYS.DDF01**, while the latter can select an application directly.

Following application selection, the terminal will issue a **GET PROCESSING OPTS** command ($CLA = 0x80$, $INS = 0xA8$) which should retrieve the Application Interchange Profile (AIP) and Application File Locator (AFL). The AIP contains information about the type of authentication supported while the AFL tells the terminal which records (list of objects) should be read from the card to perform the transaction.

Before moving on I mention that most of the information returned by the card is encoded using a format called Basic Encoding Rules - Tag Length Value (BER-TLV). In short, this format encapsulates each object in a triplet containing the tag of the object (1 or 2 bytes), the length (1 byte) and the value (a sequence of bytes representing the object).

The next step is to read the records specified in the AFL by using the **READ RECORD** command. The most important objects are the Card Risk Management Data Object List 1 (CDOL1) and the Cardholder Verification Method (CVM) List. The CDOL1 specifies which data should be included in the first **GENERATE AC** command, while the CVM list enumerates the methods that can be used to authenticate the card user (e.g. PIN or signature).

In the Chip and PIN system, as its name suggests, the most common CVM is the plain text PIN (there is an option to encipher the PIN but this is not used in the UK). The next step of the transaction is to get the PIN try counter by means of the **GET DATA** command ($CLA = 0x80$, $INS = 0xCA$). The PIN try counter represents the number of trials the card user has to correctly enter the PIN before the card becomes locked (it can then be unlocked by the issuer bank). If the counter is greater than zero the user is asked to enter the PIN number into the numerical pad of the terminal. At this point the terminal can either verify the PIN online by requesting confirmation to the issuer bank, or ask the card (offline) for confirmation. The most common method is the offline version. Thus the next step is to send a **VERIFY** command ($CLA = 0x00$, $INS = 0x20$) to the

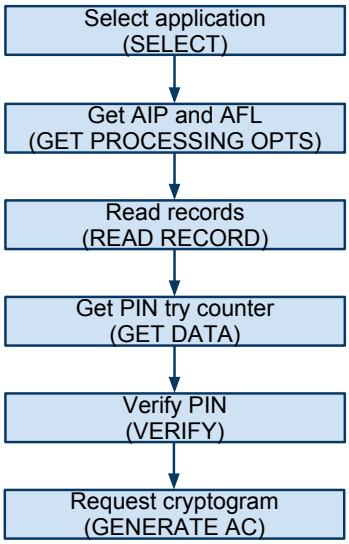


Figure 2.6: *Flow of an EMV transaction*

card with plain text PIN as the command data. If the PIN is correct the card simply replies with a success status ($SW1 = 0x90$, $SW2 = 0x00$).

The last step of a transaction is to request an application cryptogram from the card using the **GENERATE AC** command ($CLA = 0x80$, $INS = 0xAE$). This request may be for an Application Authentication Cryptogram (AAC) which requests canceling a transaction, an Authorization Request Cryptogram (ARQC) which request an online transaction (i.e. confirmation from issuer bank) or a Transaction Certificate (TC) which requests an offline transaction (i.e. no confirmation from the issuer bank). The terminal sends a list of objects as specified by CDOL1 which generally includes the transaction amount, the Terminal Verification Results (TVR - 5 bytes indicating the status of the transaction as seen from the terminal perspective), the transaction date and an unpredictable number. The card then uses its private key (only known by the card and the issuer bank) to compute a Message Authentication Code (MAC) over a set of bytes that include the data sent with the **GENERATE AC** command. If the card accepts the transaction it returns a MAC of the same type (AAC, ARQC or TC) or at a lower level (where AAC = level 1, ARQC = level 2 and TC = level 3). If the card decides to reject the transaction it will return a MAC of type AAC.

In online transactions there may be a second **GENERATE AC** command issued by the terminal where the command data is according to CDOL2 instead of CDOL1. An illustration of the entire transaction flow is presented in figure 2.6.

In the next chapters I will discuss about particular implementations of EMV, where they differ from the standard, the vulnerabilities discovered by others and my own findings during the development of the SCD.

Chapter 3

Related work

Adida et al. [1] first described relay attacks against EMV. A relay attack against the EMV system is done by forwarding the information from a genuine smartcard to another fake card. For example Alice (genuine user) goes to a shop to pay for an item of 20 pounds. At the shop, one of the attackers has tampered with the terminal. Somewhere else, at a jewelry shop, there is the attacker's partner with a fake card that communicates wirelessly with the fake terminal. When Alice enters her card and PIN number into the fake terminal the information from the original card is sent to the fake card which is in turn inserted into the original terminal at the jewelry shop.

Anderson and Bond have proposed a solution to the relay attack [2]. Their solution, called the **man-in-the-middle defense**, is a trusted device with a display that sits between the card and the terminal. This device should allow the user to verify information about the transaction such as the requested amount. Then the user could accept or decline the transaction. Building such a device has been the main target of my project.

In 2007 Drimer and Murdoch have taken a practical approach on the relay attack [9]. Using a small factor Xilinx Spartan 3 FPGA they modified a Chip and PIN terminal such that they could control the communication to the card, the PIN entry pad and the display. The modified terminal was connected to a laptop which communicated wirelessly with another laptop that in turn was connected to a fake card. They successfully tested the relay attack on a live Chip and PIN system and then proposed a solution to this attack by using a distance bound protocol. Some could argue that the system created by Drimer and Murdoch is similar to my project goals. However my goal was to create a cheap hand-held device for the purpose of showing information about the current transaction. While their system requires a terminal, an FPGA and a pair of communicating laptops, I have developed a self-containing portable solution that costs overall less than the FPGA only.

A trusted display for smartcards is already being produced by AVESO [6]. The **P300 Inlay** display is designed especially for smartcards. This display provides 6 digits via 7

segment displays and a response time below 1s. However this technology has its limitations as a solution for the man-in-the-middle defense. Firstly it needs to be built together with the card. This means that banks need to invest money in creating new cards with display, while my device could be separately purchased by interested users. Secondly it does not allow the user to accept or cancel a transaction simply because it has no input mechanism. The Emue card [11] may be a solution to this, as it provides a display and input buttons. Another solution is to use the terminal. But using the terminal key pad to allow or cancel a transaction is not possible because the transaction amount is sent after the user enters the PIN. It is not feasible to change all existing terminals or make a change in the protocol just for the sake of this application.

Murdoch et al. [22] have recently found an important vulnerability in the EMV implementation from many countries, including the UK. They have managed to complete a Chip and PIN transaction without knowledge of the user's PIN. This is a man-in-the-middle attack, where the attacker communicates different things to the terminal and card. On one side the middle-man tells the terminal that the PIN entered is correct, while on the card side the middle-man removes the PIN verification (i.e. the VERIFY command is never sent while the rest of the transaction remains unchanged). This works because the card will believe that signature authentication has been used since no PIN verification was requested. Both card and terminal keep some status bits that could be used to detect this attack but because of the complexity of the EMV implementation this mismatch is not checked. In my original MPhil project proposal submitted in January 2010, before finding about the vulnerability discovered by Murdoch et al., I had mentioned the possibility of removing the PIN verification from a transaction. Although at that time I did not think of the consequences or success of this attack (as explained by Steven Murdoch in his paper), this does not exclude that others may have found the vulnerability.

In the next chapters I describe the implementation of the SCD from a hardware and software perspective, the applications developed including the recently discovered attack, as well as some interesting details found while working on this project.

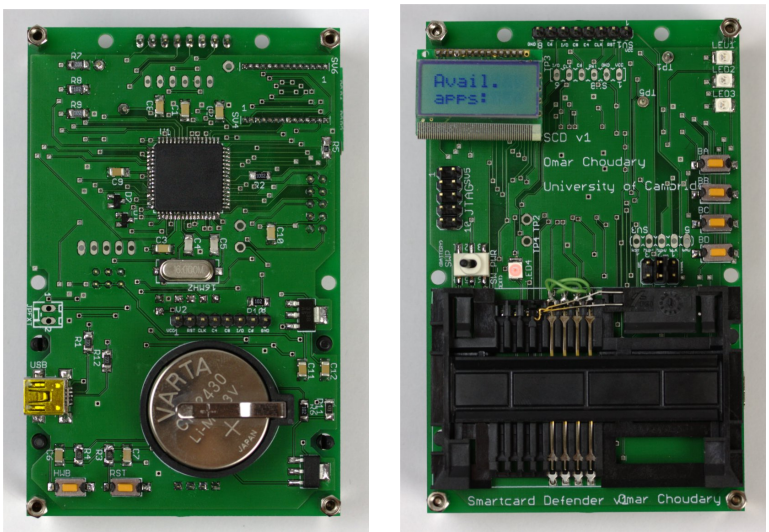
Chapter 4

SCD overview

In this chapter I present the Smart Card Detective, giving an overview of the components involved, its functionality and the applications implemented so far. Details about the implementation, the challenges encountered, and the decisions made, are described in the next chapter.

4.1 Hand-held device

An illustration of the SCD with the main components highlighted is shown in figure 4.1.



(a) (b)

Figure 4.1: *Image of the SCD, component side (a) and user interface side (b)*

The user interface is composed of an LCD with 16 characters divided in two rows, 4 push-buttons, 4 low-power LEDs, and a power switch. The LCD can be placed in two

positions, differing by a 180 degrees rotation such that the LCD screen can be read in different orientations of the SCD.

There are several power sources, which can be selected using the power switch in conjunction with an external power jumper. When the power switch is in the **BATTERY** position the SCD runs on battery power. When the switch is in the **EXTERNAL** position the power source is determined by the jumper and is expected to be 5V. The available sources then are: **TERMINAL**, which means that the SCD will be powered by the terminal; **USB**, where power is given via the USB interface; and **EXT**, where power can be given by an external source. The power switch also has a 3rd position in which no power is selected and thus the SCD in turned off.

Programming and debugging can be done using the JTAG interface. Additionally the SCD can be programmed via the USB interface using the **In-System Programming** facility which allows application code stored in the flash memory to write data into another part of the flash. The USB interface can also be used to connect the SCD to a computer and transfer data using the USB controller available on the SCD.

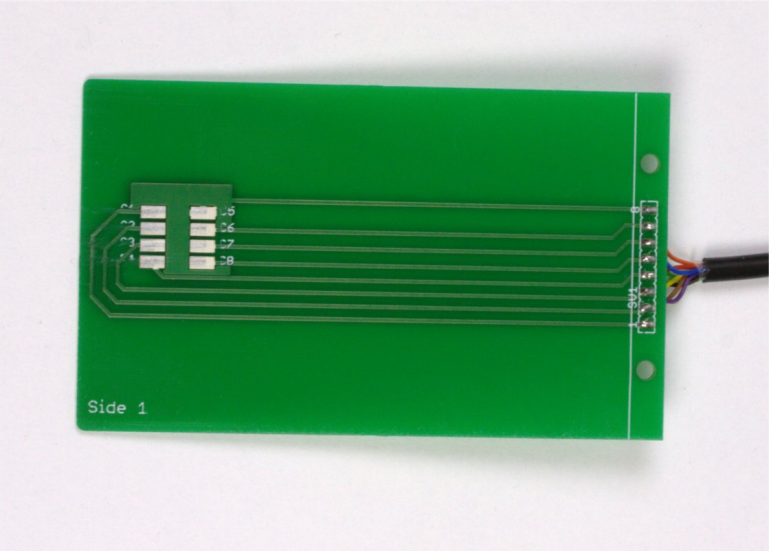


Figure 4.2: *Smartcard Interface PCB*

The user smartcard is inserted into the card slot (user interface side), and the SCD is connected to the terminal using a smartcard interface. For the smartcard interface I have designed a special Printed Circuit Board (PCB) having the same dimensions as a bank smartcard, which assures a reliable connection between the terminal and the SCD (see figure 4.2).

A normal usage scenario is as follows: first the user inserts the smartcard in the SCD, then switches on the device and selects the desired application. At this point the user inserts

the PCB card interface into the terminal and proceeds with the selected application. The applications are presented below.

4.2 Applications

There are currently five applications implemented: **Forward Commands**, **Store PIN**, **Modify PIN**, **Filter Amount** and **No PIN**.

The applications are selected from the boot menu. In order to access the boot menu the button **BB** must be pressed while resetting the device (pressing the **RST** button on the component side). Once the boot menu is accessed, the user can press **BC** to scroll through the list of applications and then select the desired application by pressing **BD**.

Forward Commands shows live information about the transaction on the LCD and logs the commands and responses without modifying or blocking the transaction. The log is stored in the EEPROM of the device and can be retrieved via any programming interface. The log mechanism can be used with all the applications presented below.

The **Store PIN** application as its name suggests is used to store a PIN into the EEPROM memory. This PIN is used by the **Modify PIN** application to modify the PIN that is transmitted in a transaction. In this way the user can make any transaction without ever typing the real PIN. To store the PIN I currently extract the necessary information from the VERIFY command. As a result the keypad of a terminal (including CAP readers) can be used to type the desired stored PIN.

Filter Amount represents the main goal of this project. This application eavesdrops on the communication between the card and the terminal until the terminal sends a **Generate AC** command. At this point the SCD blocks the communication and shows the requested amount to the user. The user can check the transaction amount by scrolling through the menu displayed pressing **BC**. Then pressing **BA** will allow the transaction to continue while pressing **BD** will terminate the transaction and restart the SCD.

Finally, the **No PIN** application is the implementation of the attack discovered recently by Steven Murdoch. Using this application the user can make a Chip and PIN transaction without knowing the PIN of a card even though the transaction receipt will read **PIN VERIFIED**.

Details about the design and implementation of the SCD are presented in the next chapter. An evaluation and analysis of the applications implemented as well as results from real tests are presented in chapter 6.

Chapter 5

SCD implementation

In this chapter I present the requirements and implementation of the SCD, the challenges encountered, and the solutions developed. This information should provide a good foundation to anyone willing to implement a similar device.

5.1 Requirements and constraints

Before choosing the hardware and software architecture I defined the specifications of the desired device in terms of functionality, user interface, cost, size, power management and software development platform.

The size was an important aspect of this project, as researchers in our department have already implemented large and hard to transport solutions [9] with similar functionality to the SCD. Thus I decided to make a portable hand-held device that any user could carry without too much burden.

The next thing to consider was the price. Since I was trying to build a portable user device the price needed to be low enough that many users could actually afford it.

The main desired functionality of the SCD was to protect the users from tampered terminals such that they could see the details of the transaction before authorizing it. I managed not only to achieve this functionality but also to implement other applications as well as the recently discovered EMV vulnerability to use a smartcard without knowledge of the PIN.

Based on the functionality requirements, I defined the basic requirements for the user interface: at least two buttons (OK, CANCEL) and an LCD display. The aim has been to keep the user interface as simple as possible while achieving the target functionality.

Power management has been a very important topic. The SCD should operate without being connected to an external power source. In such a situation there are two possible

power sources for the desired application: the power given by the terminal, or the power from a battery. The EMV standard specifies that the smartcard should consume a maximum of 80 mA which means that terminals may limit the power that is given to the SCD (which is being seen as a smartcard). Also the battery life is limited. Thus the SCD should be designed to use the lowest possible power while having enough power for the user smartcard, microcontroller and LCD.

Another two important constraints have been the frequency and maximum delay specifications of EMV. The terminal can provide a clock with any frequency in the range 1–5 MHz. Therefore the SCD can operate the user smartcard at any frequency in that range. At the same time the microcontroller of the SCD might run at its own frequency. This means that the SCD will need to ensure a correct communication with the card and terminal while dealing with three different frequency domains.

There are two delay constraints imposed by EMV. Firstly, any character sent by the card should be sent within a maximum delay called **Work Waiting Time** (WWT) from the previous character sent either by the card or terminal. The default value of the WWT is 9600 ETUs. Secondly, the card should be able to receive two consecutive characters from the terminal with a minimum delay of 11.8 + N ETUs (where N is given in byte TC1 of ATR) between the start bits of the two characters. Also the terminal should be able to receive two consecutive characters from the card with a minimum delay of 12 ETUs. Since the SCD will act as a terminal in the communication with the user smartcard, and as a card in the communication with the real terminal, it must satisfy all these constraints.

On the software side I targeted a solution that could be easily used and extended by others. My aim has been to design a device that could be actually used in practice but also by any researcher that needs an accessible open platform for EMV.

In the following sections I describe the implementation of the SCD, explaining in detail the solutions to all the constraints mentioned above.

5.2 Hardware

I have considered five different technologies for the hardware platform: small form factor FPGAs from Opal Kelly [24], powerful computer-on-module boards from Gumstix [18], development boards from XMOS [29], and 8-bit AVR microcontroller from ATMEL [4].

The Opal Kelly boards start from £120 and feature a Spartan-3 FPGA that can be clocked from 1 MHz up to 150 Mhz, an USB port, 4 push buttons, 8 LEDs, and 86 I/O pins accessible through 0.1” headers.

The Gumstix boards feature a 600 MHz ARM CPU, Bluetooth and 802.11 wireless chips, 256 MB of memory, and many I/O pins for external connections at the price of £140.

The XMOS development board starts from £80 and provides a proprietary hardware multi-threaded CPU, input buttons, LEDs, USB, 4 MB of memory, and 64 I/O pins.

All the technologies described above provide a fast and powerful prototyping framework but they are expensive even on large quantities and require additional expansion boards for custom components such as the card interface and power management circuits.

Thus I decided to make my own design using the 8-bit AVR microcontrollers from ATMEL. The AVR devices provide enough resources for my project needs, with most instructions executing in one clock cycle, a large development community, and many development tools available. This solution allows me to create a compact board with all the necessary components at a cost well below the other possibilities.

5.2.1 ATMEL AT90USB1287 AVR microcontroller

The first choice of AVR microcontroller was the **AT90SCR100**. This features two USB controllers (host and device), one smartcard interface (a hardware block dedicated to the communication with smartcards, providing an interface to transfer bytes), an AES hardware module, 64 KB of Flash and over 100 general purpose I/O pins. Unfortunately I was not able to find this controller on the market. After several requests over e-mail and phone I decided to choose another device, presented below.

The best alternative I found to the AT90SCR100 is the **AT90USB1287** microcontroller. This has many of the characteristics of the AT90SCR100 except for the smartcard interface but is highly available. Also ATMEL provides an evaluation board for this microcontroller, the **AT90USBKey**.

The most important characteristics of the AT90USB1287 for my project goals are: 128 KB of Flash (for executable code), 4 KB EEPROM (used for permanent data), 8 KB of RAM (used to store runtime information about a transaction), 4 timers (used for synchronized communication with smartcard and terminal), several sleep modes for reduced power consumption, USB controller (used as programming interface but also allows normal transfer), 16 Millions of Instructions Per Second (MIPS) at 16 MHz, and [2.7-5.5] V operation. I detail these features in the following paragraphs.

For better efficiency I decided to use a 16 MHz clock to drive the CPU. This requires a higher working voltage and power consumption than using an 8 MHz clock but allows a better sampling of the terminal clock as described below and also gives a larger processing window between data transmissions.

5.2.2 Terminal and smartcard interface

Two 8-bit timers (T0, T2) and two 16-bit timers (T1, T3) are available (please refer to figure 5.1 for the microcontroller pinout). Each timer can be configured to run from the internal clock where a divider of factor 1/8, 1/64, 1/256 and 1/1024 is available, or from an external clock. Also each timer has several comparator units that can be used to trigger interrupts, change a flag or change some of the I/O pins at any value of the timer. A special comparator unit (**A**) can be used to set the top value at which the timer will reset. These features allow the communication with the terminal at any input clock frequency and the communication with the smartcard at another independent frequency.

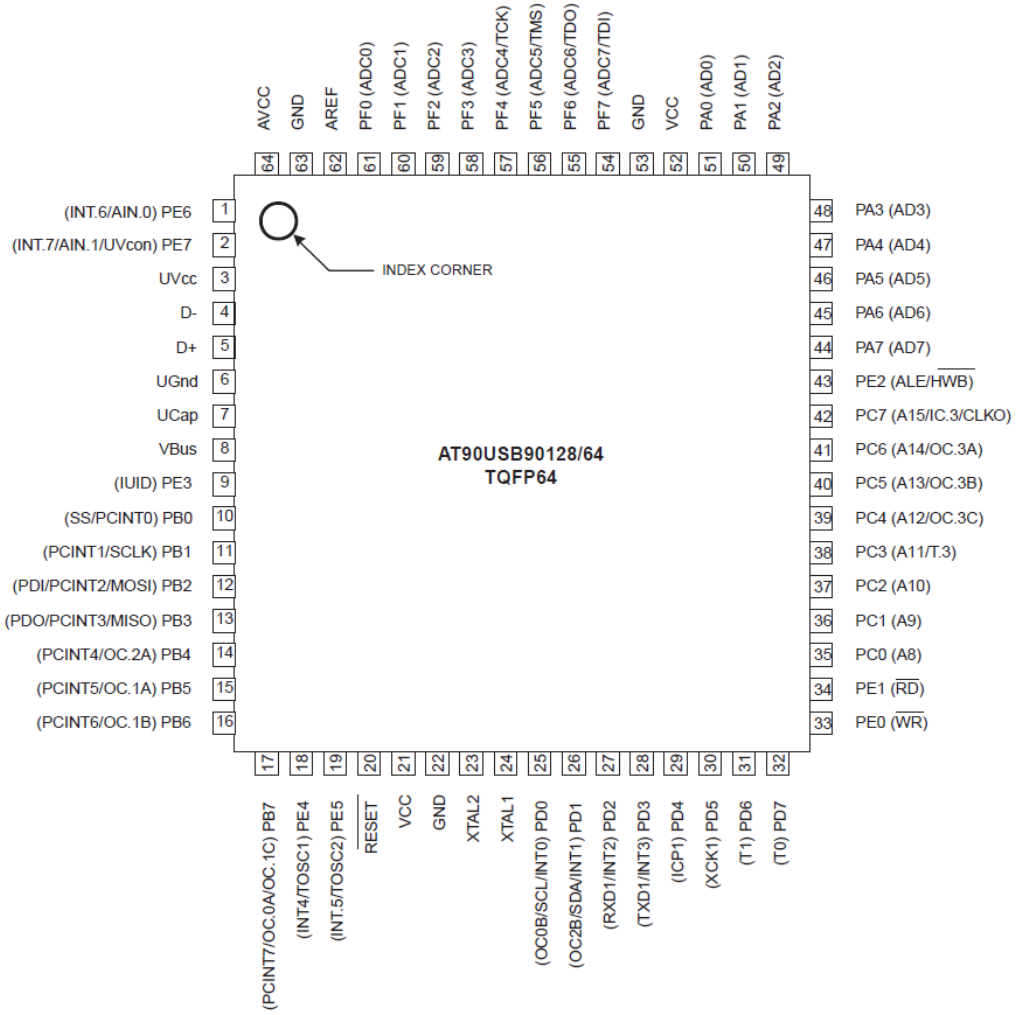


Figure 5.1: *Pinout AT90USB1287. Image from ATMEL datasheet*

To communicate with the terminal, the **CLK** signal is connected to the external input of T3 and the **I/O** is connected to **PC4** which is also one of the pins that can be changed by T3 using the output comparator unit **OC3C**. In this way the timer T3 is incremented

based on the terminal frequency. The requirements for a good sampling is that the frequency of the CPU is higher than 2.5 times the external signal. Thus, using the 16 MHz crystal it is possible to sample correctly even at the maximum terminal frequency of 5 MHz. Setting the 16-bit register OC3C to 372 (number of clocks in one ETU) it is possible to correctly transmit data by toggling the PC4 pin high or low depending on the bit that must be sent. Receiving data is done by reading the PC4 pin in the middle of a bit, by first setting OC3C to 186 (half of the ETU) and then to 372 for remaining of the bits to be received. The **RST** line is connected to **PD0** which is also the external interrupt **INT0**. This is useful because the CPU can wake up from sleep modes or can execute an interrupt subroutine based on the terminal reset signal. The external interrupt can be triggered on: any level change, low level (sustained for at least one cycle), falling edge or raising edge). Finally the **VCC** line is connected to the external power jumper **JP1**. More on this is described later under the power sources subsection. See also the complete schematic in figure 5.3.

The smartcard communication is slightly different. The smartcard clock is provided by the microcontroller using T0 and the output comparator unit **OC0A** (connected to PB7). As the CPU runs at 16 MHz, setting the OC0A register to 1 and the option to toggle the compare match output pin (**PB7**) will provide a frequency of 4MHz on that pin. The **I/O** line is connected to **PB6**, the output comparator unit **OC1B** of T1. Thus, to transmit a bit the OC1A register is set to 1488 (4 times ETU, since the CPU runs at a frequency 4 times higher than the smartcard clock) and the OC1B pin is set to change according to the bit transmitted when T1 reaches the OC1A value. Receiving a bit is done similarly but instead of changing the OC1B pin we need to check the compare match flag which should be set when T1 reaches the OC1A value. The reset signal is provided using the I/O pin **PD4**. Finally **VCC** is given with power from the main circuit using a MOSFET P-channel transistor activated by **PD7**. The I/O pins of the microcontroller allow a maximum of 40 mA per pin, while the smartcard can require up to 80 mA, thus power from the main circuit was required. The P-channel transistor was chosen because it can be opened with a ground voltage while an N-channel MOSFET requires an opening voltage larger than the output voltage (5V in this case).

5.2.3 Power sources

As described in the previous chapter there are several power sources, including battery, USB and terminal power.

The battery is needed in most practical situations, as the power from the terminal may not suffice for the entire circuit including the smartcard, or the application should run before or after the terminal has provided power. Batteries have a variable output voltage depending on the remaining life time and the current consumption. Thus a power regulator (**U2** in

figure 5.3) is needed to provide a constant voltage into the circuit. The output voltage from the power regulator should be above 4.5 V for a correct operation of the SCD. In the prototype version of the SCD I used a 9V battery which has a high capacity thus providing a good voltage for a long time. However on the PCB version I decided to put two 3V CR2430 cell batteries (for the convenience of space) which have a normal running voltage around 5V. In this situation I had to use a very low drop out power regulator combined with a good software power management. By inspecting a CAP reader from Barclays I found that it uses a similar solution, with four 1.5 V cell coin batteries instead of the two 3V batteries.

The USB is a good power source for development or research as it does not consume the battery but requires the device to be connected to a power supply such as the computer. The power from the USB is not regulated as it is assumed to be 5V, the standard bus voltage. This also allows the use of a variable power supply in order to test the circuit under different values of current and voltage.

For cases where neither USB nor battery power is available, the SCD can run with power from the terminal. Most terminals should be able to provide enough power for the SCD even though CAP readers are very limited. However the use of terminal power requires a careful design of the software as the device will have to complete any initialization routines within 80000 terminal clock cycles after which the terminal will stop the power if the ATR is not received. This may seem enough but the startup of the AT90USB1287 by itself requires 16000 clocks at a terminal frequency of 4MHz.

A last power source option is given by the pins **JPEXT1, JPEXT2**. These pins can be used to plug a source that provides 5V. Such power supply can be useful for development where a connection of this type is more convenient.

5.2.4 Peripherals

There are several peripherals available in the SCD: JTAG interface, ISP, USB, terminal connectors, smartcard interface, LEDs, an LCD and buttons.

The JTAG interface is used for programming and on-chip debug. There are several hardware tools available from ATMEL for this purpose, including the **JTAG ICE mkII**, the **AVR ONE** and the **AVR Dragon**. During the development of the SCD I have been using the AVR Dragon [5], which provides the necessary functionality at a price of £30.

For programming purposes Tuxgraphics provides a cheap and efficient ISP programmer [28] for which I included a connector in the SCD design. As well as the JTAG, the ISP interface allows to change the settings of the fuses and lock bits. The fuses are special programmable bits that control certain features of the hardware such as the initial clock divider, startup time, CPU frequency or brown out detection level. The lock bits are

similar to fuses but they control the ability to program the chip and the restrictions on different areas of the memory.

The USB port allows a fast-speed bus connection between the SCD and a host such as a PC. This connection can also be used to program the SCD by using a boot loader application (also known as **In-System Programming**). The program memory (Flash) in the AT90USB1287 is divided in two sections: **Application** and **Boot**. When the lock bits allow it, code in the Boot section can modify the contents of the Application section. Pressing **HWB** while resetting the device will make the CPU to run the code in the Boot section (the boot loader) instead of the Application section.

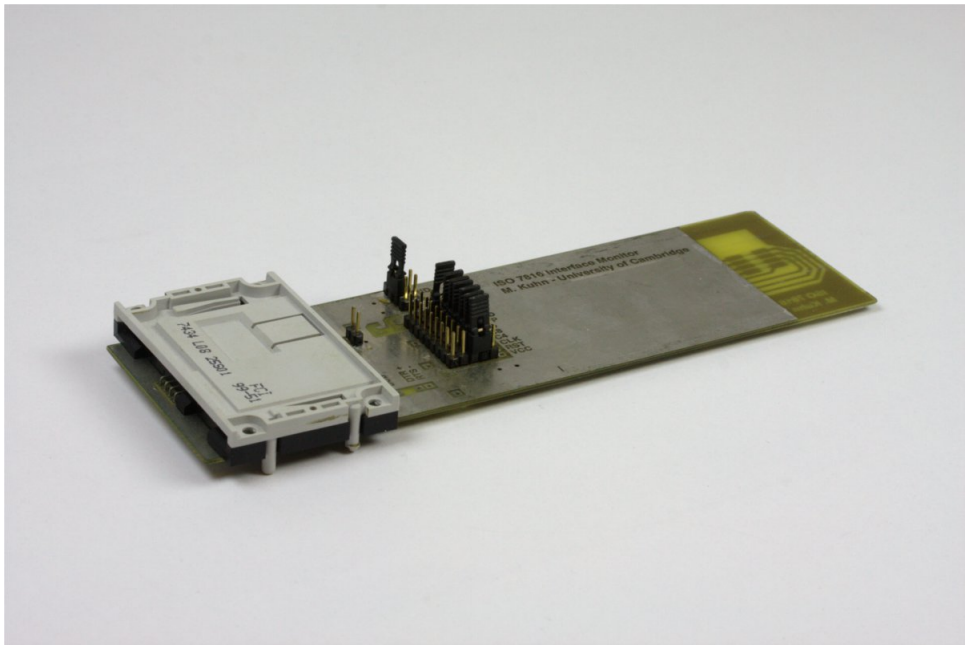


Figure 5.2: *Smartcard Interface Monitor by Markus Kuhn. Used as the card interface during development of the SCD*

In order to communicate with the terminal a card emulator board with contacts like the one shown in figure 4.2 is needed. For the connection with the card emulator the SCD has two interfaces available: an 8 pin connector (**SV1**) compatible with the **Interface Monitor V1.0** developed by Markus Kuhn (see figure 5.2), and a 6 pin connector (**SV8**). The 6 pin connector does not provide connection to the **C4** and **C8** contacts but this does not affect any current functionality as these contacts are not used in the payment system applications.

The user card is connected through a smartcard interface. This interface (**FCI** in figure 5.3) provides a physical receptacle, contacts from the card, and a card insertion switch. The card contacts are also exposed using an 8 pin connector (**SV2**), similar to the one used for the terminal connector. The card insertion switch is connected between ground

and pin **PD1** which is also the external interrupt **INT1**. As long as the card is not inserted **PD1** is tied to ground. Inserting the card will leave that pin connection open and by enabling the pull-up resistor of **PD1** the voltage will be brought to the V_{CC} level. This can be used to trigger an interrupt each time the card is inserted or removed.

The LCD has been a fundamental requirement of the SCD. Based on their popularity I decided to use Hitachi HD4478 [19] compatible LCDs. The two identical LCD interfaces (**SV4**, **SV6**) are composed of 8 data pins (**PA0** to **PA7**), 3 control pins (**PC0** to **PC2**), ground, V_{CC}, and a contrast voltage. As explained in the previous chapter I used two identical mirrored interfaces to allow the placement of the LCD in two positions. The contrast voltage should be given using a voltage divider with a variable resistor (potentiometer). From experiments with two types of LCDs I observed that the required contrast voltage for good visibility is different. The prototype board has a potentiometer which allowed me to test different values of the contrast voltage. However in the PCB version I decided to use a fixed voltage divider, targeting only the small version of the LCD, in order to keep a low cost and size of the device. The fixed value has been chosen such that the display is well readable for V_{CC} in the range [4.2, 5.2] V.

For the purpose of generality I decided to use four input buttons instead of two, and to add four LEDs that could be used to provide some information without the LCD (e.g. power on, card inserted).

The prototype and PCB versions of the SCD are presented in the next subsections.

5.2.5 Prototype

The first step in building the SCD was to create a complete schematic design that included all the components and connections in the circuit. I have used the free version of EAGLE [10] for the schematic and board (see below) design. The schematic is shown in figure 5.3.

Starting from the schematic I created a board design in order to make an approximative placement of the components before soldering them into the prototype board. The resulting board can be seen in figure 5.4.

The cost of all components (including two types of LCD and battery not shown in figure 5.4) was around £100. It required about three days of work to assemble the components and wires into the prototype board.

After fixing some power connection mistakes (which led to replacing the microcontroller), the hardware of the SCD prototype was operational.

The main problem encountered was the malfunction of the USB communication, which was tested by trying to program the chip through the USB interface. Using an oscilloscope **HP 54645D** that features two analog channels and 16 digital channels at 100 MHz I was

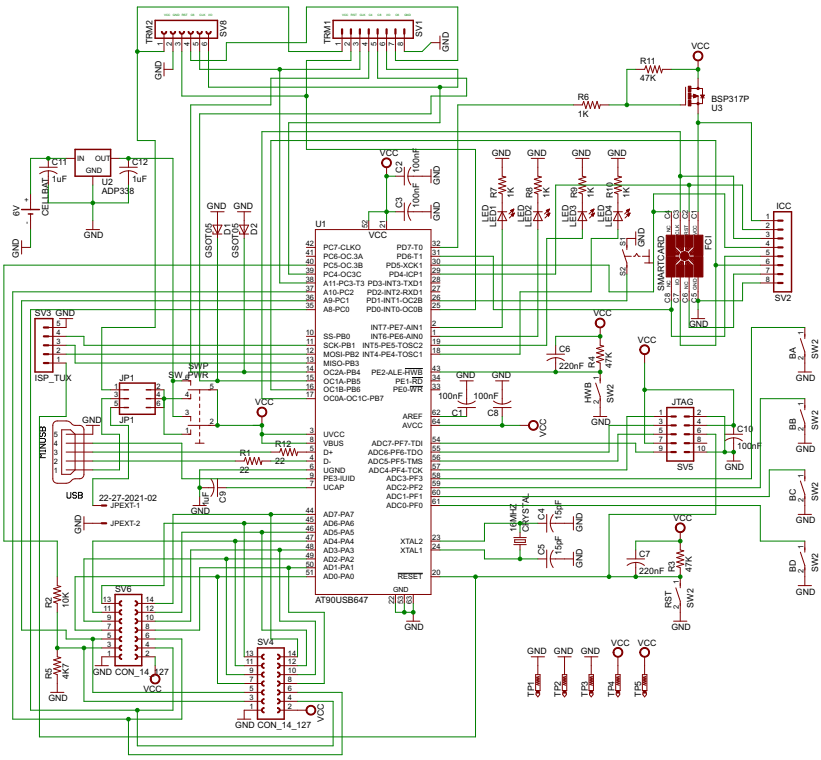


Figure 5.3: *Complete schematic of the SCD*

able to detect the problem. The USB data lines (**D+** and **D-**) should be connected with 22 Ω resistors and optionally with data line protection diodes. In the prototype version I added protection diodes for the USB data lines without considering the capacitive effect of the selected diodes. With fast-speed USB communication (about 12 Mbps) the data lines were not changing the voltage within the required time frame because of the high capacity introduced by the protection diodes. Removing the diodes completely seems a good approach in practice (as the data lines are still protected against high currents by the resistors) and was the preferred solution for the PCB design.

With the hardware in place I started to work on the software side of the project, as described in the next section. Using the oscilloscope to analyze signals and the AVR Dragon debugger to check the internal state of the microcontroller, I was able to successfully transmit sequences of bytes between the SCD, card and terminal. This proved the correctness of my design and its capability to correctly communicate with both card and terminal. Next I describe the development of the PCB version.

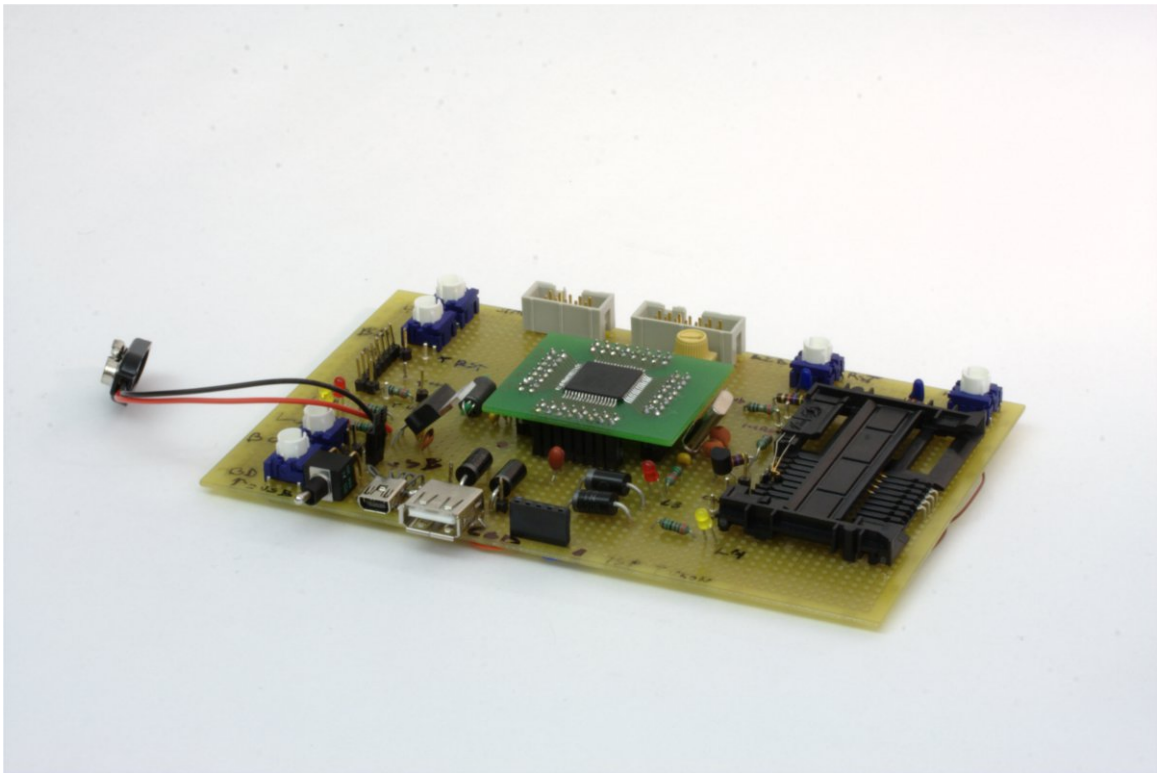


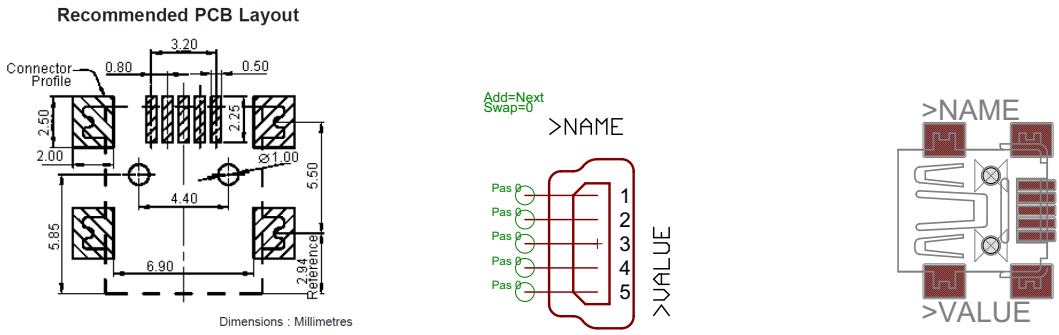
Figure 5.4: *Prototype version of the SCD*

5.2.6 PCB

Having verified the correctness of my design I started the work on the PCB version of the device.

The main part of this process involved finding the available components that would perform the required functionality and creating the schematic and board library symbols for the EAGLE board design. Most of the components have been chosen from Farnell [16] as they provide a large selection and good delivery time. I should mention that finding the right component is a matter of much experience even if it seems easy at a first glance. Even though EAGLE provides a large library of components, most of those required in my project were not already available or had different pinouts. Thus, for each component not found in the library I had to analyze its data sheet, create a schematic symbol, then create a package following the dimensions from the data sheet exactly (or else the device would not fit in the final PCB), and finally add the component to the board design. An illustration of this process can be seen in figure 5.5.

Once all the components have been placed on the board design, they must be connected using wires. Initially EAGLE provides **airwires** that connect the components in straight line. From this stage we must create non-intersecting paths which will become the visible routes in the final PCB. This process is known as **routing** and can be partially done by



(a) (b) (c)

Figure 5.5: *Process of component design in EAGLE: get component data sheet (a), create schematic symbol (b), and create board package (c)*

the EAGLE board editor using the **autorouter** utility. However in complex designs such as the SCD, the autorouter is not able to do a proper routing. Thus I routed all the paths manually.

A PCB can have multiple conductive layers, isolated from one another. This allows each layer to have its own mesh of routes, which becomes necessary where many components are connected within a small area. Some designs use two layers for the routes, and two additional layers for the ground and power planes. For my design I have used two layers with ground planes between components on both layers. This decision was partially constrained by the free version of EAGLE which allows the use of maximum two layers but also by the price of the PCB manufacture which is dependent on the number of layers required. The final PCB design is shown in figures 5.6 and 5.7.

The next step was to send the PCB design for manufacturing. There are several companies that can produce PCBs in small quantities. For the manufacture of the SCD I used **PCB Pool** [26] and for the card interface I used **PCB Train** [27]. I decided to use PCB Pool for the SCD even if they were more expensive that PCB Train, because they have a better specification of the manufacturing process, complete details of what will be included in the result (silkscreen, soldermask), a good process tracking service, and they accepted the EAGLE board design directly. Generally a design is sent for manufacturing in a standard format, known as **Gerber**. From a board design, EAGLE can easily produce Gerber files but these must meet the manufacturer criteria in terms of wire width, space between components, wires and drills, etc. As I did not have prior experience in PCB manufacturing I preferred to send the board design. In the case of the card interface (see figure 4.2) I needed a PCB less than 0.8 mm thick and PCB Pool does not produce this. Thus I created the necessary Gerber files taking as guidelines the specifications from PCB Pool and I sent the design to PCB Train which can produce 0.8 mm PCBs.

The price for one board of a similar size to the SCD, with two layers including silkscreen

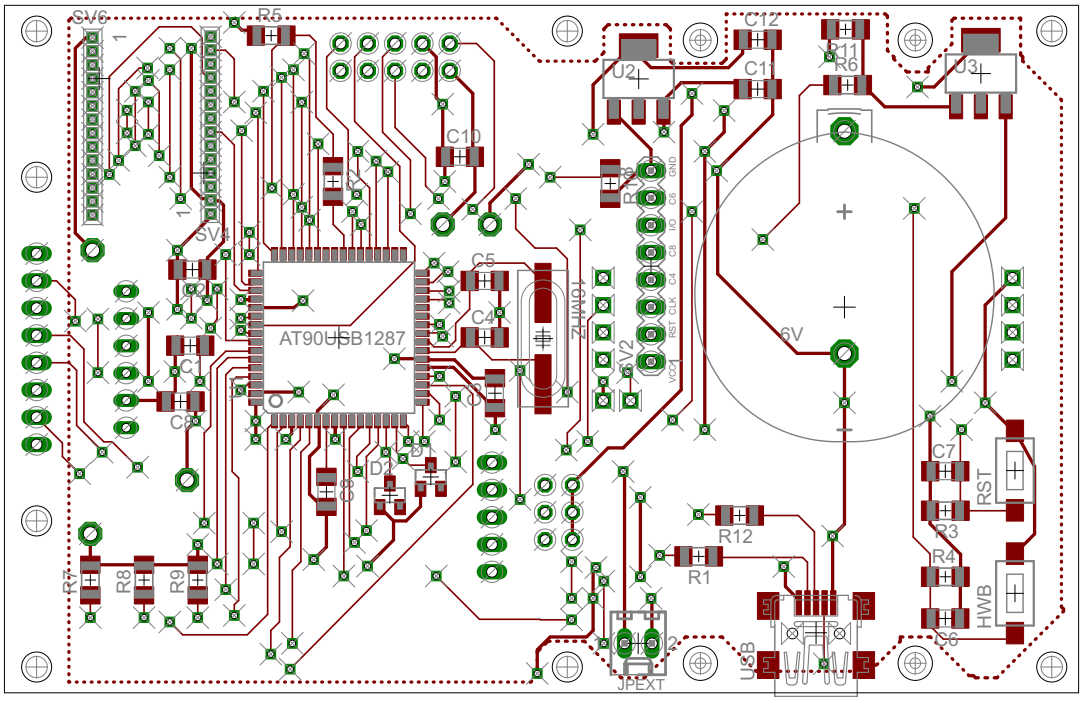


Figure 5.6: *PCB design top side*

and soldermask on both sides is £60 at PCB Pool and £30 at PCB Train. The price decreases with the number of boards ordered, down to £3 per PCB when 100 boards are ordered at PCB Train. If the SCD is built in large quantities (more than 1000) the expected price (including components) would be around £20.

The final step has been to solder the components on the board. Illustrations of the result are shown in figure 4.1. A few things went wrong due to errors in the board design. The most impacting error has been inversing the smartcard interface contacts in one of the sides. The solution has been to cut the contacts and add wires in place as it can be observed in figure 4.1. A second error was caused by an improper value of the voltage divider used for the LCD contrast voltage which made the display invisible for voltages close to 4.5V. This was corrected by a simple replacement of a resistor. Other minor errors related to the bad alignment of the text which was caused by having used a smaller font size than the one permitted by the manufacturing process.

In the following section I describe the software architecture of the SCD, while the evaluation is presented in the next chapter.

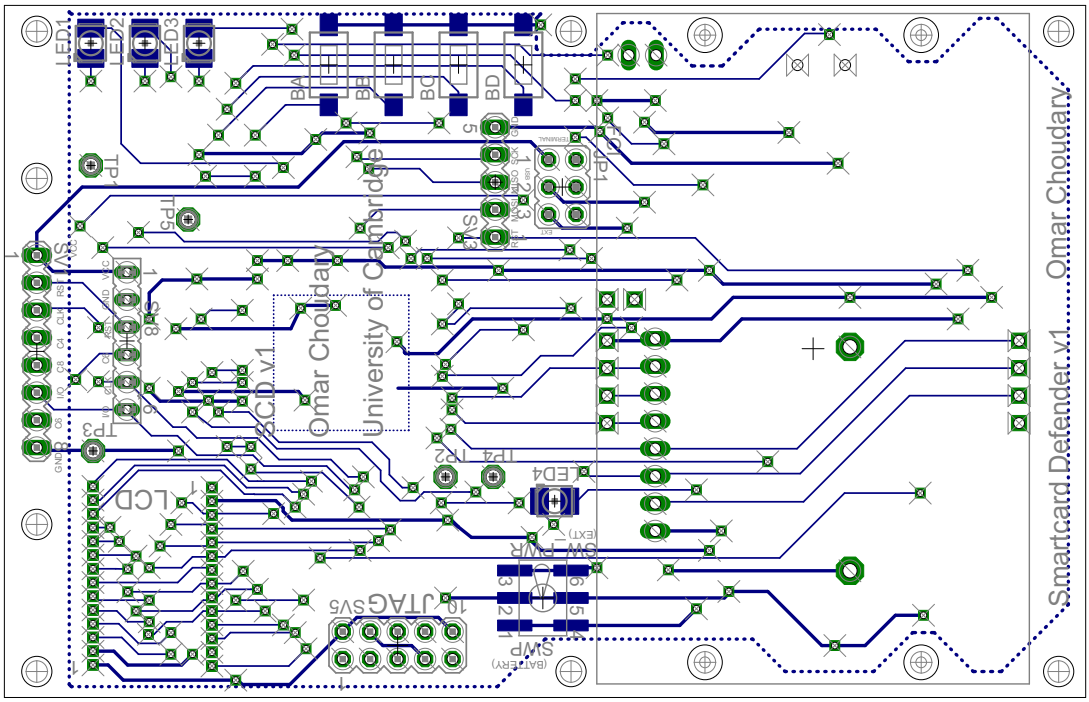


Figure 5.7: *PCB design bottom side*

5.3 Software

Atmel provides a good free development environment (IDE), called **AVR Studio**. This IDE provides the integration of a text editor, memory, registry and I/O viewer, chip programmer, and C and assembler compiler. Combined with an on-chip-debugger, the **AVR Studio** allows step by step execution and variable examination, either directly or by means of the assembler code.

I decided to write the software mostly in C with some small parts written in assembler. The C language offers a higher level of programming which facilitates code development, verification and management, while the assembler allows a strict execution timing regardless of the compiler.

AVR-GCC [7] is a free C compiler for Atmel AVR microcontrollers. It is based on the well-known **GCC** compiler but has the required extensions for the AVR instructions. Most of the memory management including the heap allocator and malloc routines are especially designed for microcontrollers with small memory (usually below 8KB SRAM). The compiler is well maintained and there is a large community of users. The AVR-GCC has become so popular that it has been included as the main C compiler of AVR Studio.

5.3.1 Architecture

I have designed the software in a hierarchical manner so that only some low-level files need to be rewritten for different microcontrollers. The overall structure is shown in figure 5.8 and is explained below.

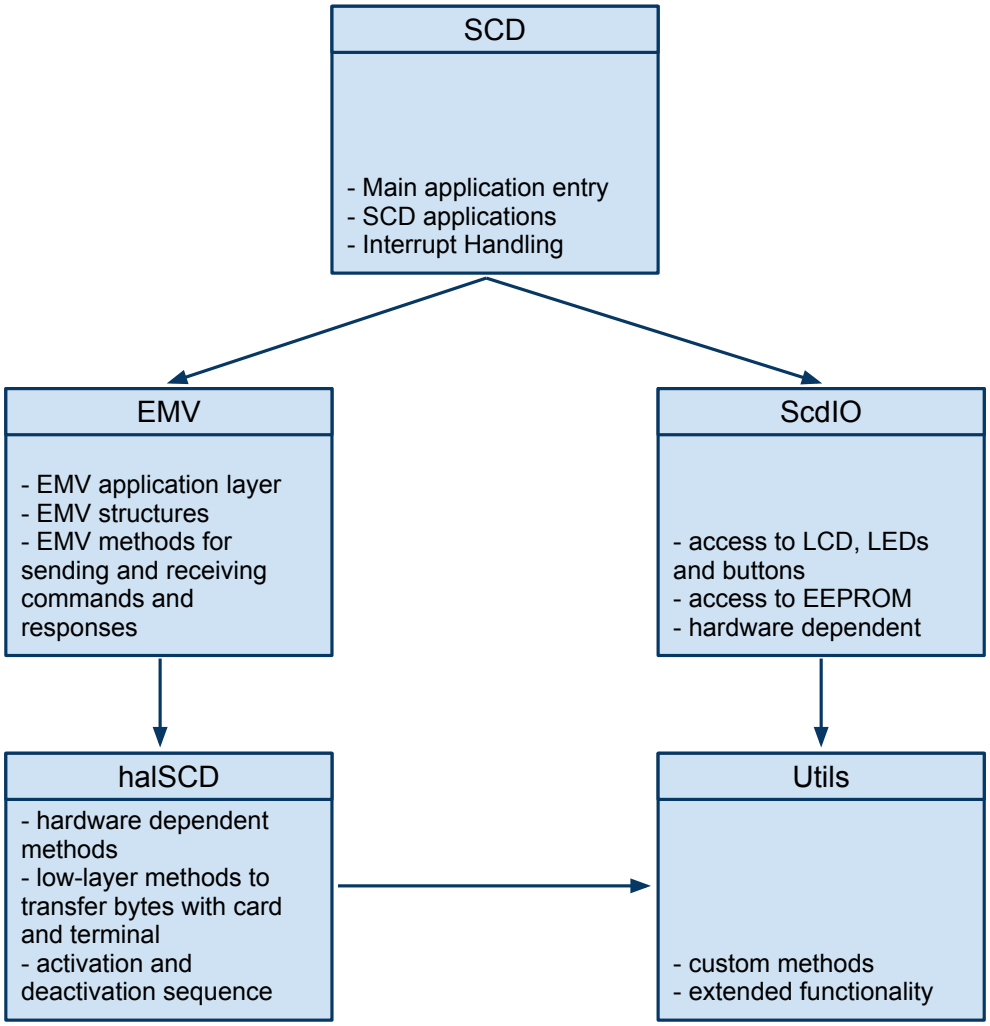


Figure 5.8: *Software architecture of the SCD*

The **halSCD** is the hardware abstraction layer. This block contains the implementation of the low level functions of the EMV protocol, such as activation and deactivation sequences, send and receive bytes, parity checking and retransmission, and sending and receiving the ATR. The **halSCD.h** file contains the definition of methods needed by higher level functions (see below) while the **halSCD.c** file contains the actual implementation. Thus, in order to use the software on another microcontroller only the C file has to be rewritten.

The **EMV** block as it names suggests contains all the structures and functions needed for the EMV protocol. The important structures defined are: **CAPDU**, **RAPDU**, **RECORD** (for the list of BER-TLV objects returned in READ RECORD commands) and **TLV** (a BER-TLV object). The functions defined are used for sending and receiving commands and responses from card and terminal, and for parsing records in order to identify important information such as the transaction amount. All the functions defined in **EMV.h** and implemented in **EMV.c** rely only on the functions from halSCD and thus can be safely used with any other platform as long as the halSCD methods are correctly implemented for that platform.

All the peripherals access is done through the **ScdIO**. The methods defined in **ScdIO.h** provide an easy access to the LCD, LEDs, buttons and EEPROM. As with the hardware abstraction layer, the implementation provided in **ScdIO.c** is hardware dependent. Thus for an implementation on another platform the C file needs to be rewritten.

The **Utils** block defines some general methods that are used frequently such as accessing 16 bit data in an atomic manner. The code in these methods should be portable across all the AVR microcontrollers.

Finally the **SCD** is the core of the solution. It uses all the software components in order to implement the applications described in the previous chapter. The **SCD.h** defines important parameters such as application ID, and the applications available. The implementation of those applications are found in the **SCD.c** file, which also handles the interrupts and power management. Unfortunately as interrupts and sleep modes are dependent on the microcontroller some parts of this file should be rewritten for a different architecture, although most of the code for the applications will be the same.

In order to comment the code I decided to use Doxygen [8]. Doxygen provides a good framework to document and publish source code. The only requirement to use Doxygen is to write the comments of the source code in one of the three formats accepted. Using a configuration file it is possible to specify which parts of the code should be made available together with the documentation, what files to include in the documentation, and the output desired: PDF, HTML or LaTeX. The complete source code and documentation in HTML format is available on my website [23]. However, this code is currently available for evaluation purposes only. I do not give the right to use this code in any commercial or non-commercial purposes.

Probably the most interesting parts of the software are the transmission of bytes to both terminal and card as they use different operating frequencies, and the initialization of the communication as the protocol requires the bytes in the ATR to be transmitted soon after power is given from the terminal.

The basic concept behind byte transmission with both card and terminal has been explained in the previous section. I use the timers available to send data at the correct

frequency. The code for sending and receiving bytes is shown in Appendix A.

5.3.2 Initialization sequence

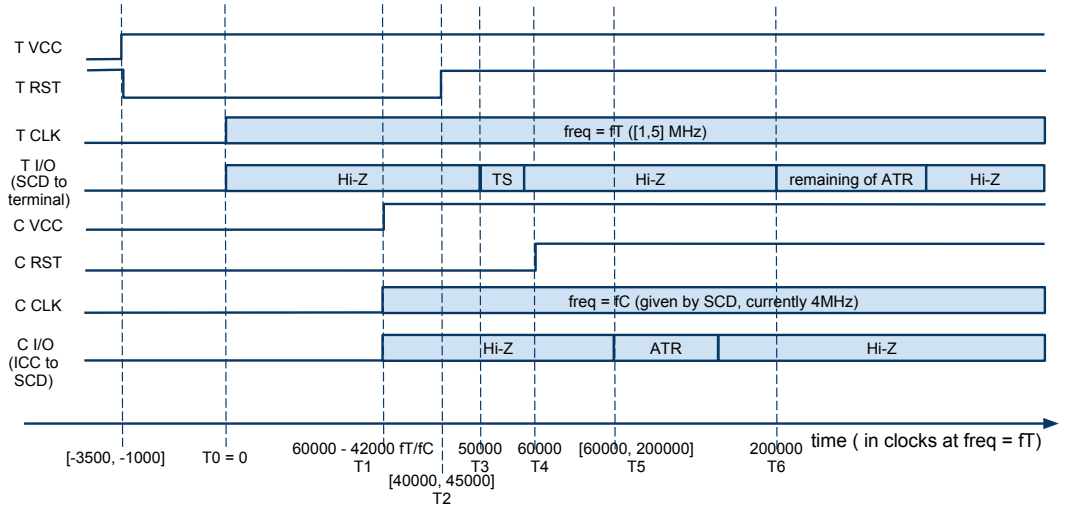


Figure 5.9: *SCD initialization phase*

The initialization phase is shown in figure 5.9. I have designed this process to be independent from the terminal frequency (works with any terminal frequency in the range [1,5] MHz). The **T** signals refer to the terminal and the **C** signals refer to the card. The **X** axis shows the time elapsed in terms of clock cycles at the terminal frequency. As described in the background section, the initialization procedure has a strict timing. The terminal provides the voltage and clock (at time T0 in the figure) after which it sets the reset line to high, between 40000 and 50000 clock cycles (T2). Within 42000 clock cycles from T2 the card must respond with the first byte of the ATR. It is important to notice that only the first byte (TS) is required within the 42000 clock cycles to continue with the transaction. Actually by intercepting the communication with a genuine card I have noticed an important delay between this first byte and the remaining of the ATR. The maximum delay between successive bytes (including those in the ATR) is 9600 ETUs which is equivalent to 3571200 terminal clock cycles (or 890 ms at a terminal frequency of 4 MHz).

The first thing to notice is that the SCD provides the voltage and clock to the card (at time T1) depending on the fraction between the terminal clock frequency (**fT**) and the card clock frequency (**fC** - provided by the SCD). T1 is chosen such that after 42000 clock cycles at fC (time T4) the reset line from the terminal has been already set to high (at time T2) and the first byte of the ATR (TS) has been sent to the terminal (at

Table 5.1: Initialization sequence for the communication between terminal, SCD and card

Time	Action
T0	Terminal provides clock
T1	SCD provides clock to card
T2	Terminal changes reset line to high
T3	SCD sends byte TS of ATR to terminal
T4	SCD sets card reset line to high
T5	Card returns the ATR
T6	SCD sends remaining bytes of ATR to terminal

T3). Thus at T4 the SCD can correctly set the card reset line to high and expect the ATR from the card at any moment (T5) between T4 and T4 + 45000 clock cycles at fC. The communication between the terminal and SCD will remain in a correct active state because the byte TS has already been sent, which gives a working window of 3571200 terminal clock cycles as mentioned above. After receiving the ATR from the card, the SCD can send the rest of the ATR bytes to the terminal. At this point the SCD is ready to receive the first command from the terminal and the card is ready to receive the first command from the SCD. This procedure is summarized in table 5.1.

It is important to mention, that if the SCD requires more processing time between the bytes sent by the terminal, this additional time can be requested during the initialization procedure. The byte TC1 of the ATR tells the terminal the amount of extra time (in ETUs) to be added between consecutive bytes sent. Normally a card would set this byte to 0 in order to minimize the delay of a transaction but I successfully used different values with CAP readers.

5.3.3 Interrupts and power down modes

The AT90USB1287 microcontroller has many internal and external interrupts. The internal interrupts are caused by internal events such as timer overflows, analog to digital conversions, or watchdog overflow, while the external interrupts are caused by the change in level of pins **INT0,..,INT7** and **PCINT0,..,PCINT7**. The difference between the INTX and PCINTX interrupts is that each of the former category has a dedicated interrupt vector (and consequently a particular handling routine) while for the latter any change in a pin PCINT0 through PCINT7 that has the interrupt enabled will trigger the same interrupt (**PCI0**).

Any interrupt has an associated interrupt vector, and if a handling routine is defined this

will be executed when the interrupt occurs. Interrupts are allowed by setting a global interrupt flag in the status register (**SREG**) and each particular interrupt is enabled by setting an interrupt enable bit. Interrupts can be used also to wake up the microcontroller from a sleep mode. This is used in the SCD to wake up the microcontroller when the terminal provides the clock. First the timer T3 (connected to the terminal clock) is set to trigger an interrupt when a certain value is reached. Then the microcontroller is put in sleep mode. Only when the terminal provides clock the SCD wakes up and initiates the communication. This allows an important power saving.

There are 5 different sleep modes available in the AT90USB1287: idle, power down, power save, standby, and extended standby. They differ in the parts of the hardware that remain active during sleep and the time required to resume normal state. Power down consumes the least energy but also requires the longest period to recover (more than 4 ms). On the other side, idle provides a fair amount of energy saving while resuming activity in only 6 clock cycles. This is because power down stops the main PLL clock completely while the idle mode only stops the CPU clock. I decided to use the idle mode also because it is the only sleep mode that allows the use of timer overflow interrupts to wake up the microcontroller, which is needed by the SCD operation.

5.3.4 Memory

The AT90USB1287 provides 3 types of memory: Flash, SRAM and EEPROM.

The Flash has 128 KBytes and is used to store the application code. As described previously this memory is divided in two sections, the application and the boot section. Both can be used for executing code. In normal operation the execution will start from the application section but the user can also select execution from the boot section.

The SRAM provides 8 KBytes of data space that can be accessed in two clock cycles. Together with the memory allocator provided with AVR-GCC this space becomes very useful for storing dynamic data such as transaction information in the case of the SCD.

Finally, the EEPROM provides 4 KBytes of permanent storage. This space is essential for storing data that must remain in memory even after the SCD is powered off or restarted. In the SCD, the EEPROM is used mainly to store transaction logs, the selected application, a transaction counter and a custom PIN.

5.3.5 Operation

Having defined the main characteristics of the SCD and the software architecture I will now describe the overall operation of the device. All the methods referred below are implemented in the file **SCD.c**.

There are many applications implemented in the SCD, but the overall execution flow is the same. The steps presented below assume that the SCD has just been powered up or restarted.

First an initialization routine (**InitSCD**) is called in order to set up the pins correctly (input/output, low/high), enable any necessary interrupts, and retrieve data from the EEPROM.

Then the SCD checks if the **BB** button is pressed. If so, a menu is shown on the LCD, that allows the user to select the desired application (see the previous section). If the button is not pressed, the current application is selected based on the data from the EEPROM. In the case of no application previously chosen (empty data in EEPROM), a default application is used.

With the exception of the **EraseEEPROM** (which erases the EEPROM and then restarts the SCD), all the applications involve communication with the card and the terminal. Therefore, the next step is to put the SCD in sleep mode in order to save power and wait for terminal clock.

Once the terminal is connected (by inserting the card interface into the terminal) and has provided clock, the SCD wakes up. Before executing the selected application the watchdog timer (**WDT**) is enabled. The WDT is used to reset the device after a given time out to prevent dead locks or unexpected loop execution.

All the applications start by initializing the communication with the card and the terminal as described previously. Hence the user's card must to be inserted into the card slot before starting the application. Then the external interrupt INT0, which corresponds to the terminal reset line, is enabled. This is necessary in order to reset the SCD in case the terminal issues a reset or ends the transaction (in both cases the reset line will toggle from high to low).

The applications will have a loop where they transfer commands and responses between the terminal and the card. As they do so, the applications call the **wdr** instruction used to reset the WDT and avoid a system reset (which will happen if any application loops indefinitely). The **wdr** instruction is called between commands or responses which ensures that this instruction is only called when the device is transferring data correctly. It is not a good idea to call the **wdr** at the end of a transaction because the WDT will most probably timeout. This is because the maximum timeout is 4 s while a transaction can have a much longer duration including PIN entry and any other user input.

The end of an application is expected from the terminal connection. That is, the SCD expects the terminal reset line to go low and then execute the interrupt routine for INT0. The reset line must go low at some point either because the transaction has ended or because the card interface has been removed from the terminal.

When the INT0 interrupt routine is executed the SCD saves any transaction data as required and then restarts. Saving transaction data cannot be done after reset because the contents of the SRAM memory will be erased. Thus the INT0 handling routine provides a good place to save transaction data into the EEPROM.

After use the transaction information recorded in the EEPROM can be easily transferred to a PC using any of the programming connections available: USB, ISP or JTAG. I have designed the current software such that it will record transactions linearly, using memory in an efficient manner.

In the current implementation the EEPROM can store information about up to three transactions, each having 30 command-response pairs. However I have used simple but inefficient command and response delimiters (stream **CCCCCCCCCC** for commands and **AAAAAAAAAA** for responses). Using a more efficient coding might provide space for saving an extra transaction.

5.4 Terminal emulator

During the development of the SCD I needed a terminal emulator device. CAP readers provide a good interface but they are limited to the applications already installed in the device (generally **IDENTIFY** and **SIGN**). Therefore I decided to build my own terminal emulator.

Figure 5.10: *GemTwin USB smartcard reader*

In order to obtain a physical connection I used an USB smartcard reader from Gemalto [25] (see figure 5.10). This provides the same interface as a real terminal but allows a PC to act as the terminal software.

I have written the software for the terminal emulator in **C#**. The Windows platform provides an API to communicate with USB smartcard readers. This API has been used

in a C# application to extract data from SIM cards [17]. I built my own terminal emulator on top of the available code, which provided an interface to communicate with the Smartcard API but no EMV functionality. Thus I was able to transmit commands and receive responses through the USB smartcard reader. In this way I could write my own transaction flow, sending commands and analyzing responses as needed.

The complete software for the terminal emulator is available for evaluation.

In the next chapter I describe the evaluation of the SCD, analyzing the hardware and software design, as well as the overall functionality.

Chapter 6

Evaluation

With the hardware and software in place it was time to perform an evaluation of the SCD, in terms of hardware, software, and functionality. The results and observations are presented below.

Most of the tests have contributed to progressively improve the device in terms of performance and functionality. During the laboratory experiments I used the three analysis tools presented earlier: the .NET debugger to trace commands and responses as seen by the terminal emulator, the AVR dragon to check the state of the microcontroller (in terms of memory, execution path and I/O levels), and the oscilloscope to trace the signals from card and terminal (amplitude, frequency, bit duration).

6.1 Basic functionality

One of the first things to test was the correct functionality of the peripherals: the LCD, buttons and LEDs. This was done in a method (**TestHardware**) that performs simple I/O operations.

The next step was to verify the correct transmission of bytes between the terminal and SCD. This was done first by means of the terminal emulator as I could create my own test case and verify the results. Initially I verified that the ATR is correctly sent by the SCD, and then I checked that sequences of commands are correctly received and responded to. After that I created a reliability test case, where I wanted to verify the correct transmission of a large number of bytes. For this scope I created a loop transaction flow using the terminal emulator. A similar loop was used in the SCD. The SCD successfully exchanged data uninterrupted for 30 min (approximately 10 MB of data at a terminal frequency of 4 MHz, considering data is sent half of the time) before I ended the test. Similar tests have been done to check the correct communication between the SCD and card.

Once the SCD proved to work with the terminal emulator, I started the tests with CAP readers, which provide a real test scenario. I have used three different CAP readers, from Vasco, Natwest and Barclays. They execute a real transaction flow and have different working characteristics. The Vasco reader operates at 1 MHz, while the Natwest and Barclays readers run at 1.5 MHz. Also the Vasco reader starts a transaction but stops the power and clock soon after receiving the ATR. The transaction is then restarted when the user selects the desired application. The Natwest reader keeps the power and clock from the moment the card is inserted until the transaction is over. Finally the Barclays reader behaves similarly to the Vasco reader but sends an initial short restart signal before actually starting operation. The diversity of operation provided by these three readers provided a good input to improve the SCD functionality. The evaluation of the complete SCD functionality including tests on CAP readers is discussed in section 6.3.

6.2 Power consumption

The main objective of the SCD was to be used as a hand-held device in real payment scenarios. Thus power consumption is a critical factor as the battery may be exhausted very quickly if good power management is not in place.

Using a variable power supply I tested the SCD operation with different voltage and current intensity. I found several factors that make a big difference in the overall power consumption, as presented below.

First of all, it is important the way in which the I/O line is set up for reception mode. The pull-up resistor of the I/O pin (**PB6** for the card) should be enabled instead of driving the line high. When the line is driven high the power consumption of the ICC is much higher (with spikes of about 30 mA difference) than when using the pull-up resistor. I have noticed an important difference by changing the sequence of the following instructions, which set the card I/O line to reception mode (state Z):

- Good way:

```
DDRB &= ~(_BV(PB6));
PORTB |= _BV(PB6);
```

- Bad way:

```
PORTB |= _BV(PB6);
DDRB &= ~(_BV(PB6));
```

There are six I/O ports (labeled A through F) available in the AT90USB1287. Most of the ports have 8 pins. The direction (input or output) of each pin is controlled by

Table 6.1: Port pin configurations for AT90USB1287. Extracted from datahseet

DDRxn	PORTxn	I/O	Pull-up	Comment
0	0	Input	No	Tri-state (Hi-Z)
0	1	Input	Yes	Pxn will source current if ext. pulled low
1	0	Output	No	Output Low (Sink)
1	1	Output	No	Output High (Source)

the **DDRx** registers (where bit 3 of DDRB controls the pin **PB3** of port B), while the **PORTx** registers control the voltage on that pin (high or low). The possible states for each port are shown in table 6.1.

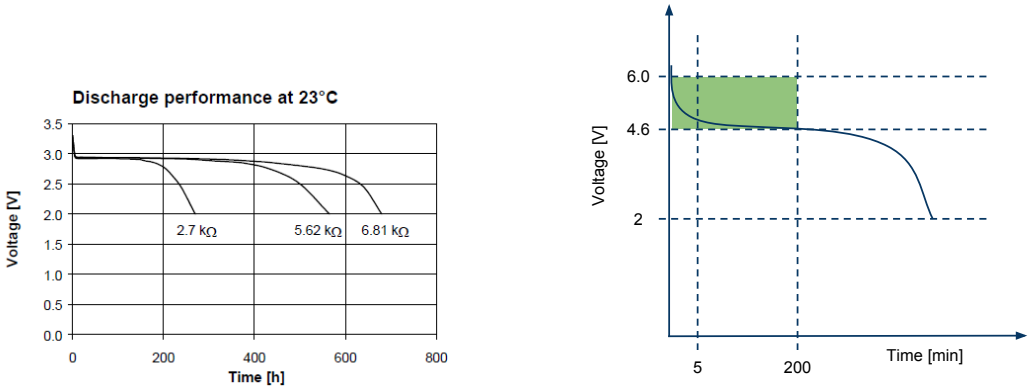
Back to setting the I/O line correctly, what happens in the first case is that initially PB6 is set as input with no pull-up (assuming PORTB6 was 0), and then PORTB6 is set to 1 which enables the pull-up. This sequence will cause the voltage on the I/O line to gradually increase from 0 to approximately 3.78 V which is given by the pull-ups of the AT90USB1287 and ICC. In the second case, the I/O line is first set to high and then DDRB6 is set to 0, changing the pin direction to input which enables the pull-up resistor. In this case the voltage on the I/O line spikes to around 5 V and then decreases slowly to the value of 3.78.

I also noticed that the ICC does not drive the I/O line correctly (after testing with several cards). When the ICC transmits a **ZERO** bit to the SCD, it pulls the line low (ground) correctly. However, when the card transmits a **ONE** bit, it first uses the pull-up resistor to rise the voltage to around 4 V and then immediately switches the port to high impedance such that the voltage drops to around 1.5 V. Thus, if the SCD uses the high impedance mode on the I/O line as well, the communication ends. So the terminal (and implicitly the SCD) must enable the pull-up resistor on the I/O line even when it is receiving a byte from the ICC. Such functionality does not respect the EMV standard which states that the voltage used by the card for a transmitted bit **ONE** should be [3.5, 5] V for V_{CC} = 5V.

Another important power consumer is the LCD. The model I use (EVERBOUQUET MC0802A-SGR) consumes between 10-30 mA during operation. There are other LCDs with lower power consumption, but this was chosen based on its low cost (£5). In this situation is very important to use the LCD only when needed. After many experiments I realized that the best method to reduce the LCD power consumption is to delay sending any commands to the LCD until the last possible moment. Although the HD4778 controller (available in the MC0802A LCD) provides commands for turning the LCD on and off, these do not change the power consumption. In fact, sending a power off command (with or without prior initialization) to the LCD will increase the power consumption.

Resetting the SCD after using the LCD seems to be the best solution to keep a low power consumption. This will restart the LCD controller, which will not start running (and thus will not consume power) until a first command is sent.

With all the improvements in place, the total consumption of the SCD (including LCD and card operation) is around 40 mA.



(a) (b)

Figure 6.1: *Battery discharge for a CR2430 cell battery under low current consumption (less than 2 mA) (a), and estimated discharge for two CR2430 batteries under higher consumption (average of 30 mA)*

The battery discharge in time for one of the 3V CR2430 cell batteries is shown in figure 6.1(a). As it can be seen, this type of batteries are designed for a load of at least 2.5 K Ω (i.e. less than 2 mA at 5V). In full operation the SCD consumes up to 40 mA, while in sleep mode it consumes less than 20 mA. Considering an average consumption of about 30 mA, and based on several measurements of the battery voltage during operation, I approximated the discharge of two CR2430 batteries under continuous use of the SCD as shown in figure 6.1(b) (the green area represents the time during which the two batteries provide enough voltage for correct operation of the SCD).

6.3 Functionality tests

After checking (and fixing problems as necessary) the basic functionality and performance of the SCD, I went on to verify the applications described in chapter 4.

The first series of tests have been done on CAP readers, as they perform a transaction with all the necessary steps for testing my applications, including PIN verification (VERIFY command) and amount authorization request (GENERATE AC command).

Table 6.2: Log of CAP transaction

Command	**Response**	**Details**
00A4040007 A0000002440010		SELECT
	6A82	file not found
00A4040007 A0000000038002		SELECT
	6A82	file not found
00A4040007 A0000000048002		SELECT
	9000 + FCI data	Selection OK
80A8000002 8300		GET PROCESSING OPTS
	9000 + AIP + AFL	OK
00B2010C00		READ RECORD
	9000 + CDOL2 + CVM + CDOL1	Selection OK
80CA9F1700		GET Data (PIN try counter)
	9000 9F170103	OK, 3 retries
0020008008 24XXXXFFFFFFFFFF		VERIFY, PIN=XXXX
	9000	OK
80AE80002B + data		GENERATE AC (ARQC)
	9000 + CID + ATC + cryptogram + IAD	OK
80AE000011 + data		GENERATE AC (AAC)
	9000 + AAC data	OK

The SCD successfully executes all applications (**Forward Commands**, **Modify PIN**, **Filter Amount**, and **No PIN**) on the Vasco and Natwest CAP readers (see figure 6.2). Using the log functionality with the **Forward Commands** application I captured the information of a CAP-Identify transaction, where the user enters the PIN and gets a secure code for online authentication. This transaction is shown in table 6.2, where non-essential information that could be used for identification has been marked with **X** or omitted. The transaction flow is the same for the **Modify PIN** and **No PIN** applications, while for **Filter Amount** the transaction might end after the VERIFY command if the user does not accept to continue.

Based on the log from the CAP transaction, I observed two important differences from the standard EMV specification. Firstly, the CAP application does not use the traditional **SELECT 1PAY.SYS.DDF01** command to start the transaction, but the selection by Application Identifier (AID). As can be seen from the log, the CAP reader issues multiple SELECT commands (with different AIDs) before finding the correct CAP application on the card (AID = A0000000048002 in this case). This suggests that the CAP reader has several applications installed, possibly for different cards. Secondly, after using the AID selection, the CAP reader does not issue a final SELECT command before continuing with the transaction, as it is specified in the EMV standard.

The final test consisted in verifying the correct operation of the SCD with a real terminal, completing an online transaction. For this purposes I have asked permission to use the

Figure 6.2: ***Forward Commands*** *application tested on Natwest CAP reader. The SCD has blocked the transaction after the PIN has been entered and is waiting for the user to select if the transaction should continue (yes) or not (no)*

SCD at the cafeteria in our department. After a few failed attempts, the SCD has successfully executed the **Filter Amount** and **No PIN** applications. The unsuccessful attempts were caused by an incorrect implementation of TLV object decoding. Using the logs from the failed attempts I was able to replicate the live transaction on the terminal emulator and thus find and correct the issue.

I was expecting the **Filter Amount** application to work correctly since I tested the functionality on CAP readers and terminal emulator. However I was surprised to see that the **No PIN** application succeeded, which shows the vulnerability has not been fixed yet.

Recently, a journalist from TAC Presse (France) has come to our department to make a reportage about the vulnerabilities of Chip and PIN cards.

First Steven Murdoch has helped us in setting up the relay attack, where a fake terminal

shows a transaction for £5.00 but actually requests a payment authorization for £123.45. Using the SCD between a real card and the fake terminal we were able to see the correct amount (£123.45) on the display and cancel the transaction (see figure 6.3).

(a) (b)

Figure 6.3: *SCD operating as trusted display in a relay attack scenario: demonstration system (a), close up on fake terminal and SCD displays (b). Images offered by Steven Murdoch*

Then we have used the SCD to perform the **No PIN** attack at the local cafeteria and even in some random shops in Cambridge. We have successfully bought books and DVDs worth over £50 at one of the shops using the journalist's card but typing PIN **0000**. Even more, we have performed the tests without warning and nobody has noticed the hidden device or fake card (the card interface connected to the SCD). After the transaction we have disclosed the attack to the shop manager who said that such attacks occur very often. The manager mentioned that during busy periods like Christmas credit card frauds occur at least once a week. Because shops cannot longer check the cards (as the current policy is to let the customer handle the card insertion and removal) the criminals can use fake cards and devices similar to the SCD to perform fraud.

Chapter 7

Conclusion

In this thesis I have presented my work for the MPhil project in Advanced Computer Science. My work has involved many different tasks: designing the schematic and board circuits, creating a prototype, sending the PCB to manufacturing and assembling components, developing the entire software for the device, debugging hardware and software, testing the entire solution with CAP readers and my own terminal emulator, tests with live terminals within the department, and real tests with journalists in the city. I have performed all the work and I managed to create a working device within the less than 5 months duration of the MPhil project.

I have built a hand-held device, called **Smart Card Detective** (SCD), that can protect smartcard users from several attacks, but can also showcase vulnerabilities in the Chip and PIN system. This device contains an ATMEL AVR AT90USB1287 microcontroller that mediates the communication between a smartcard and a terminal, buttons, LEDs and an LCD. The cost of the device has been around £100 (including PCB manufacturing), and in large quantities the expected price is below £20.

Using the SCD I developed the **Filter Amount** application, which was the main goal of the project. This application eavesdrops on a transaction and blocks a payment authorization request until the user verifies the correctness of the transaction. The user is able to check the transaction amount on the LCD and then decide if the transaction should continue or not.

Additionally I have developed a **Modify PIN** application which replaces the PIN entered on a terminal by a PIN stored in the SCD memory. The main utility of this application is that users do not have to disclose the real PIN and thus can avoid situations where the PIN is seen by criminals looking over the shoulder. There are important security issues with this approach (if the device is stolen then the PIN is useless), but the objective here was to test such functionality.

Steven Murdoch et al. have recently discovered an important vulnerability in the Chip

and PIN system where a PIN transaction can succeed without entering the correct PIN although the receipt will read **PIN VERIFIED**. I have implemented this **No PIN** attack on the SCD with just minor modifications to the **Modify PIN** application, which shows the flexibility and potential of the device that I created.

All the applications have been successfully tested on a terminal emulator, CAP readers and live terminals. Steven Murdoch has kindly helped by preparing a relay attack scenario to test the SCD. Using the **Filter Amount** application we correctly identified the amount mismatch. Also, we have tested the **No PIN** attack on a live terminal at the local cafeteria. Even more, I have conducted real tests at random shops in Cambridge and the SCD was able to exploit the PIN vulnerability.

The commercial interest of such device is uncertain. Although such a device can be very useful, carrying yet another gadget every time you go shopping is at least inconvenient. Also the current version of the SCD requires a wired connection between the device itself and the card interface that is inserted into the terminal. However, there are some practical uses of such a device: a user attorney for making high-amount transactions such as buying a car, a research platform for EMV, testing equipment for payment system developers to verify the correct functionality of cards and terminals.

One of the future developments of the SCD might be to remove the wired interface between the device and the card interface. One possibility is to add a wireless chip into the card interface that would communicate with the SCD. An practical improvement might be to create a styled plastic finish similar to the CAP readers, so that users could be more tempted to use the device.

Some companies such as Emue already produce credit cards with an integrated LCD and buttons. Such cards can also provide a trusted display for smartcard users. However they are limited to one card per display. Thus the card issuer would need to invest in every card, while a device like the SCD may be used with any of the existing cards and given only to interested clients.

Based on the experiments described in this thesis we can observe that several vulnerabilities remain in the payment system. Probably more will show up as banks introduce the contactless and mobile payment solutions. Even though banks have the first call to fix existing vulnerabilities or create better security devices, costs and reputation stand in the way. In such scenario the device I have created, the SCD, can help users avoiding fraud, and can also help to discover and fix any remaining vulnerabilities.

References

[1] Ben Adida, Mike Bond, Jolyon Clulow, Amerson Lin, Steven Murdoch, and Ron Rivest. Phish and Chips (Traditional and New Recipes for Attacking EMV). In *Cambridge Security Protocols Workshop*, 2006.

[2] Ross Anderson and Mike Bond. The Man-in-the-Middle Defence. In *Cambridge Security Protocols Workshop*, 2006.

[3] APACS. 2008 fraud figures announced by APACS. `http://www.ukpayments.org.uk/media_centre/press_releases/-/page/685/`.

[4] Atmel. 8-bit AVR MCUs. `http://www.atmel.com`.

[5] Atmel. AVR Dragon. `http://support.atmel.no/knowledgebase/avrstudiohelp/mergedProjects/AVRDragon/AVRDragon.htm`.

[6] AVESO displays. `http://www.avesodisplays.com`.

[7] AVR Libc. GCC compiler for AVR microcontrollers. `http://www.nongnu.org/avr-libc/`.

[8] Doxygen. `http://www.doxygen.org/`.

[9] Saar Drimer and Steven J. Murdoch. Keep your enemies close: distance bounding against smartcard relay attacks. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.

[10] EAGLE Layout Editor. `https://www.cadsoft.de`.

[11] Emue. Emue Card, a credit card with embedded keypad, display and microprocessor. `http://www.emue.com/site/home.htm`.

[12] EMVCo. *EMV Book 1: Application Independent ICC to Terminal Interface Requirements, Version 4.2*, June 2008.

[13] EMVCo. *EMV Book 2: Security and Key Management, Version 4.2*, June 2008.

[14] EMVCo. *EMV Book 3: Application Specification, Version 4.2*, June 2008.

[15] EMVCo. *EMV Book 4: Cardholder, Attendant, and Acquirer Interface Requirements, Version 4.2*, June 2008.

[16] Farnell UK. `http://www.farnell.co.uk`.

[17] Gemalto. GemPC Twin. `http://www.gemalto.com/readers/`.

[18] Gumstix. Overo Air. `http://www.gumstix.com`.

[19] HITACHI. Dot Matrix Liquid Crystal Display Controller/Driver. `http://www.sparkfun.com/datasheets/LCD/HD44780.pdf`.

[20] ISO/IEC 7816. *Integrated circuit(s) with contacts*.

[21] MasterCard International. Chip Authentication Program - Functional Architecture. Available upon request.

[22] Steven J. Murdoch, Saar Drimer, Ross J. Anderson, and Mike Bond. Chip and pin is broken. In *IEEE Security and Privacy Symposium*, 2010.

[23] Omar Choudary. Complete source code for the Smart Card Detective. `http://www.cl.cam.ac.uk/~osc22/scd/html/`.

[24] Opal Kelly. Overo Air. `http://www.opalkelly.com/`.

[25] Orouit. A Smart Card Framework for .NET. `http://www.codeproject.com/KB/smart/smartcardapi.aspx`.

[26] PCB Pool. `http://www.pcb-pool.com/`.

[27] PCB Train. `http://www.pcbtrain.co.uk`.

[28] Tuxgraphics. ISP Programmer. `http://tuxgraphics.org/electronics/200901/tuxgraphics-isp-header.shtml`.

[29] XMOS. XC-1A development kit. `https://www.xmos.com/products/development-kits/xc-1a-development-kit`.

Appendix A

Source code for byte transmission

```
/**
 * Sends a byte to the terminal without parity error
 * retransmission
 *
 * @param byte byte to be sent
 * @param inverse_convention different than 0 if inverse
 * convention is to be used
 *
 * The terminal clock counter must be started before
 * calling this function
 */
void SendByteTerminalNoParity(uint8_t byte,
        uint8_t inverse_convention)
{
        uint8_t bitval, i, parity;
        volatile uint8_t tmp;

        // check we have clock from terminal to avoid damage
        // assuming the counter is started
        if(!GetTerminalFreq())
                return;

        // this code is needed to be sure that the I/O line
        // will not toggle to low when we set DDRC4 as output
        TCCR3A = 0x0C;              // Set OC3C on compare

        PORTC |= _BV(PC4);          // Put to high
```

```
DDRC |= _BV(PC4);          // Set PC4 (OC3C) as output
Write16bitRegister(&OCR3A, ETU_TERMINAL); // set ETU
Write16bitRegister(&TCNT3, 1);   // TCNT3 = 1
TIFR3 |= _BV(OCF3A);       // Reset OCR3A compare flag

// send each bit using OC3C (connected to the
// terminal I/O line each TCCR3A value will be visible
// after the next compare match

// start bit
TCCR3A = 0x08;

// while sending the start bit convert the byte if
// necessary to match inverse conversion
if(inverse_convention)
{
        tmp = ~byte;
        byte = 0;
        for(i = 0; i < 8; i++)
        {
                bitval = tmp & _BV((7-i));
                if(bitval) byte = byte | _BV(i);
        }
}


while(bit_is_clear(TIFR3, OCF3A));
TIFR3 |= _BV(OCF3A);

// byte value
parity = 0;
for(i = 0; i < 8; i++)
{
        bitval = (uint8_t) (byte & (uint8_t)(1 << i));

        if(bitval != 0)
        {
                TCCR3A = 0x0C;
                if(!inverse_convention)
```

```
                        parity = parity ^ 1;
                }
                else
                {
                        TCCR3A = 0x08;
                        if(inverse_convention)
                                parity = parity ^ 1;
                }

                while(bit_is_clear(TIFR3, OCF3A));
                TIFR3 |= _BV(OCF3A);
        }

        // parity bit
        if((!inverse_convention && parity != 0) ||
                (inverse_convention && parity == 0))
                TCCR3A = 0x0C;
        else
                TCCR3A = 0x08;

        // wait for the last bit to be sent (need to
        // toggle and keep for ETU_TERMINAL clocks)
        while(bit_is_clear(TIFR3, OCF3A));
        TIFR3 |= _BV(OCF3A);
        while(bit_is_clear(TIFR3, OCF3A));
        TIFR3 |= _BV(OCF3A);

        // reset OC3C and put I/O to high (input)
        TCCR3A = 0x0C;                  // set OC3C to 1
        DDRC &= ~(_BV(PC4));
        PORTC |= _BV(PC4);
}
```