Växjö University

School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

SCOS
Smart Card Operating System

Jonas Fredriksson
Mikael Wahlberg

Nov
2002

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 02141
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--02141/--SE

1 Introduction

This report presents our graduation project for the bachelor degree in computer science at the school of mathematics and systems engineering at Växjö University. The project was handed out to us by the department of information logistics at Strålfors AB, in Ljungby. We have, after analysing the problem statement received from Strålfors, developed an application with the provisional title: ”SCOS”, Smart Card Operating System.

Henrik Gunnarsson and Kim Paulsson at Strålfors and Thomas Ratier and Neil Mckeeney at Atmel provided us with expert advice. Our instructor was Morgan Ericsson at the school of mathematics and systems engineering.

1.1 Background

Memory cards without a processor are often used for simple applications when data should be protected but no computation is needed before presentation. These cards are cheaper than processor-based cards and therefore popular in many applications. One drawback is that they do not communicate with the same protocol as processor-based smart cards, and therefore not always can be used in the same card reader. One other drawback is that the security level of memory cards often is poorer than the security level of process based cards.

Because of the decrease in bandwidth in semiconductors, the memory capacity is increasing, and the processor use up less space on the card. You could with these facts in mind expect that processor-based cards in the future will be more frequently used even in cost sensitive applications. Strålfors makes smart cards, with and without processors. Today, when they are making process based cards, they licence operating systems for these cards from extern contractors. If they would have access to their own operating system it would give them new opportunities to offer customer based solutions.

The development of software for smart cards is something that will grow a lot in the nearest future. Today we use theses cards in cell phones, as cash cards and some car manufacturers use them as a combined alarm and start key. In Sweden we have had our ”cash cards” for some years now and it takes time for people to accept these new techniques, especially when it comes to how they shall do with their money, cash or card, or both at the same time? But as soon as they accept these new techniques it will probably explode with new smart card solutions on the market. Then it will be more and more important for the companies that develop the software to have an operating system that gives them the possibility to “personalize” the cards, and that is when it could be a great advantage to have an operating system that is developed by, and for, themselves. They can build it to support functions that are important to the company.

1.2 Problem discussion

Strålfors had several projects that we could choose from, but it was the problem statement for an operating system for smart cards that caught our attention. It was an area that we knew very little about and the curiosity we felt made our choice of project pretty easy.

This is the problem statement we got from Strålfors:

> ***"Develop a simple operating system for a process based smart card. The operating system should be able to protect data with PIN codes. Functions meant for creating files, reading and writing to these files, and protection of their information with different keys for reading and updating respectively. The means for encryption of the information with any of the crypto algorithms DES, RSA or elliptic curves ought to be implemented."***

Even though the problem statement we got from Strålfors clearly specifies what should be done, we chose to formulate our own problem statement. We did that because we wanted to work according to a less concrete statement, which allowed us to do a much more extensive problem analysis in this report.

Our modified statement of the given problem is:

> ***"Find out if a basic operating system, made according to the ISO/IEC 7816-4 standard, for a process based smart card can be developed within a short period of time, with reasonable difficulty and at a low cost using the developing environment provided by ATMEL. Analyse in what areas the use of crypto can be useful in the OS, but also examine in what field of application a separate smart card crypto function could be used."***

1.2.1 Raised questions

The problem statement that we formulated above raised many questions, that we have tried to answer. We feel that it could be useful to present them. Of course, our expectation is that much more questions will be answered in this report, but here are the most significant questions we identified when we analysed the problem statement:

- What parts are fundamental to implement in order to make a smart card OS operational?
- What knowledge is mandatory to make the development of an OS feasible?
- What pre-made parts exist and which of them can be used?
- How extensive is the process of implementing, for the OS, mandatory crypto? Could the implementation of a separate crypto function be meaningful and if so in what areas?

1.3 Constraints

In order to make this project practicable we had to introduce some limitations. Only basic commands in conjunction with a simple file structure and file management procedures will be implemented in the prototype. The file system need not be proven the most effective, but nevertheless provide the mandatory services. The main purpose is to provide an OS that is serviceable but not optimised.

We will in this report discuss those parts of the OS that we ourselves have implemented, but not the code for I/O management and other parts, which were provided by Atmel. Only mandatory information about these parts will be discussed. There might also be necessary information left out in this report due to Atmels wish of confidentiality.

We will not implement the part regarding cryptography that is mentioned in the problem statement. Despite this fact we will mention it in our report. We have added Appendix A to the report covering most of the, for this report important, theory regarding cryptography.

1.4 Method

We were more or less forced to work on this project in an explorative manner. The reason for this was the lack of problem domain understanding. This made the project a whole lot more exciting.

We started the project by reading and getting familiar with the ISO/IEC standard and the different commands it specified. To gain further understanding of the given problem, we tried to communicate with a already finished application using the commands specified in the standard.

Our next step was to analyse the development process applied in the environment we should use in the project. We read manuals and worked according to different tutorials. Then to confirm our gained knowledge as well as broaden it we made a trip to Atmel's factory in Rousset, France. There we also gained knowledge about what pre-made application parts could be used in our project.

Back home we began the implementation in an explorative way, starting by designing a basic file system. When the file system was done, we could initiate the implementation of ISO specified commands. During the continuing of our project we wrote down important notes useful for the creation of this report.

1.5 Requirement specifications

Our original requirements specification is very basic because of the simple fact that we were not completely sure at what it should contain. Also the standards we followed when implementing SCOS filled most of the purposes a requirement specification is supposed to fill. Nevertheless, we have written a simple requirements specification stating the essential parts needed to make an operating system functional.

1.5.1 Essential parts

- **Interpreter**: To communicate with the Operating System we need to implement an interpreter. This component should be able to receive a command interpret it and trigger essential actions and coordinate other components. The interpreter should consists of:
    - *Input buffer*
    - *Buffer with available commands*
    - *I/O components (already implemented by ATMEL)*
    - *Comparison and selection*
- **File system**: To manage all the data we need a file system. It should be able to create files, store them in the memory and also keeping track of where the file is stored. We should be able to retrieve specific data using this file system. The file system should consist of:
    - *Memory management*
    - *A file structure*
    - *File management (for example create and select files)*
    - *File referencing*
- **Security architecture**: To fulfill the basic security requirements we need to implement security architecture. This component should contain the different security levels of the card and also keep track of the current state of security. The security architecture should consist of:
    - *A security ladder (defining the different levels of security)*
    - *Current state variable (what security level is currently satisfied?)*

1.6 How the report is outlined

This is a short summary of the report that we have done in our project for the bachelor degree.

In the first chapter we have an introduction. In this introduction we have a discussion around the problem and how we interpreted it. We also explain what a smart card is and what they can be used for and why they shall be used. We explain questions that was raised while trying to understand the problem. After these questions we tell you which constraints we put on the project. The constraints will also be motivated. The last part in the first chapter is how we worked, the works procedure.

The second chapter is about the standard that worked as a detailed requirements specification to us in our work. This chapter is meant to provide you with the necessary knowledge about this domain. The standard, which is international, is called ISO/IEC 7816 and certain important parts in this standard are described in this chapter. We describe the data structures, i.e. the commands that you use to communicate with the operating system on the microprocessor. The card security architecture is also explained here. This part contains security status, security attributes and security mechanisms. We describe the APDU message structure, which a smart card OS uses to communicate with the hardware. We tell you about the coding conventions for command headers, data fields and response trailers.

The third chapter contains an explanation about the development environment. We have used the AT90SC development kit from Atmel, also called the SDK. The software we used was the AT90S IAR C compiler and C-Spy from IAR Systems. After this we describe a little about the AT90SC micro controller, which is the one on the smart card.

The fourth chapter of this report is a work analysis. This chapter is the core of our report, where we define exactly what we implemented when developing the SCOS, how we did and also try to motivate why we did it. We explain our method, our groundwork and the filestructure we implemented.

In the fifth chapter we will present an example of a possible “real world” usage of SCOS. We think that this will help to provide greater understanding of what SCOS really is.

The sixth chapter is about conclusions, where we look forward and tries to see what can be done in the future. How can we make the things we have done even better? What can be done when it comes to new functions and higher security?

At the end of this report we have references, definitions and abbreviations and finally appendix.

2 Theory

This chapter is meant to provide you with the necessary knowledge about the standard that we had to follow. It is important to view this part of the report as a knowledge provider and not as a specification of our implemented work. The specification of the implementation will be discussed in a later chapter.

2.1 International standard

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. In the field of information technology, ISO and IEC have established a joint committee, ISO/IEC JTC 1.

International Standard ISO/IEC 7816-4 was prepared by Joint Technical Committee ISO/IEC JTC 1, Information technology[11]. This part of ISO/IEC 7816 is one of a series of standards describing the parameters for integrated circuit cards with contacts and the use of such cards for international interchange. These cards are identification cards intended for information exchange negotiated between the “outside world” and the integrated circuit in the card. Because of an information exchange, the card delivers information (computation results, stored data), and/or modifies its content (data storage, event memorization).

The ISO/IEC 7816-4 standard, [11], has guided us through the development process of the SCOS application. A short description of the standard and its most important guidelines, in this report, therefore feels necessary. For more detailed information about the standard we refer to the ISO/IEC 7816-4 specification[11]. Again, keep in mind, while reading this section, that this is a short summary of the international standard and not a specification of our work. That is, everything that has been written in this section is not actually implemented in our application.

2.2 Scope

The part of ISO/IEC 7816 that we will describe here will cover:

- The structure and foundation of commands and responses, transmitted by the interface to the card.
- The structure of files and data, as seen at the interface when processing inter-industry commands for interchange.
- Access methods to files and data in the card.

2.3 Data structures

This section contains information on the logical structure of data as seen at the interface, when processing inter-industry commands for interchange, i.e. the commands that you use to communicate with the operating system on the microprocessor.

2.3.1 File organization

There are two categories of files supported by the standard. We have the Dedicated File (DF) and the Elementary File (EF). DF's can be seen as directory files containing EF's or additional DF's. An EF could not contain other files, only data, and a DF could not contain data, other than a file identifier and references to the files it accommodates.

The DF at the root of the structural file hierarchy is mandatory and is called master file, MF (see Figure 2.1). The lower level DF's is optional.

Two types of EF are defined.

- Internal EF: Those EF are intended for storing data interpreted by the card, i.e., data analysed and used by the card for management and control purposes.
- Working EF: Those EF are intended for storing data not interpreted by the card, i.e., data to be used by the outside world exclusively.

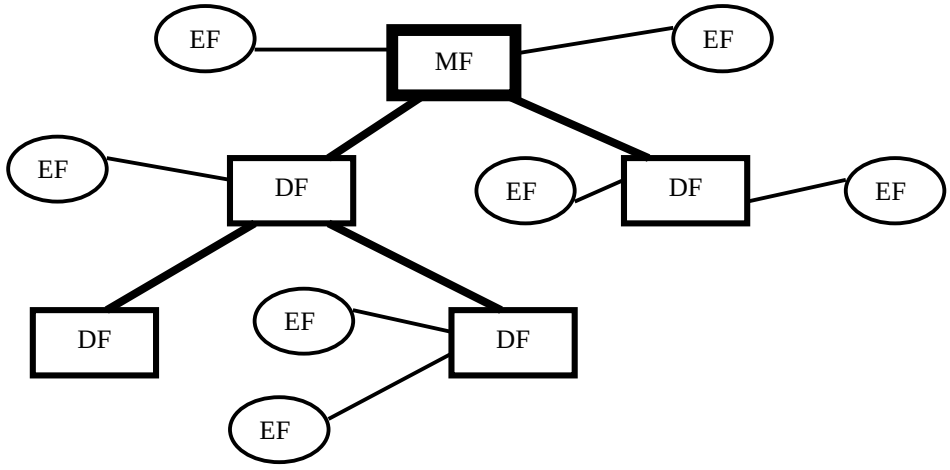

Figure 2.1 Logical file organization (example).

2.4 File referencing methods

To navigate through any file system, the file system in the SCOS application included, you need at least one file referencing method. According to the standard at least one of the following file referencing methods have to be implemented:

- Referencing by file identifier: Any file may be referenced with a file identifier coded on 2 bytes.
- Referencing by path: Any file may be referenced with a path (concatenation of file identifiers). The path begins with the current DF and ends with the file that was to be referenced. Between these two paths, any successive parent DF's is specified.
- Referencing by short EF identifier: Any EF may be referenced by a short EF identifier coded on five bits valued in the range from 1 to 30.
- Referencing by DF name: Any DF may be referenced by a DF name coded on 1 to 16 bytes. The DF name must be unique.

2.4.1 Elementary file structures

There are several different structures for data storage defined in the standard. The most common is the transparent file structure in which the EF is seen at the interface as a sequence of data units. The other variant is the record structure in which the EF is a sequence of individually identifiable records. The record structural method provides three different types, linear EF with records of fixed size, linear EF with records of variable size and cyclic EF with records of fixed size. There is a major difference between the two linear types and the cyclic one. One example of the cyclic type, is the phonebook in a cell-phone. When you reach the last record and move one step further, you come to the first record again. This is not the case in the linear types, where it stops when you reach the last record. Figure 2.2 shows a principal sketch of the four EF data structures.

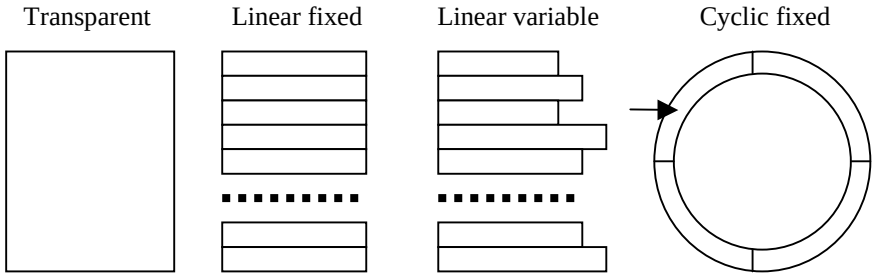

Figure 2.2 EF structures[11].

2.4.2 Data unit referencing

Within each EF of transparent structure, each data unit can be referenced by an offset. By default, i.e., if the card gives no indication, the size of the data unit is one byte.

2.5 Card security architecture

The security benefits are one of the reasons why smart cards are used in many different areas. To be able to provide this security the cards have to have security architecture. Security status, security attributes and security mechanisms are parts of this architecture and these we will try to describe in this section. Security attributes are compared with security status to execute commands and/or to access files.

2.5.1 Security status

Security status represents the current state. This state is possibly achieved after completion of answer to reset (ATR) and possible protocol type selection (PTS) and/or a single command or sequence of commands, possibly performing authentication procedures. To simplify this means that when a security key has been verified corresponding status byte change state and indicates that the key has been verified.

The security status may also result from the completion of security procedure related to the identification of the involved entities, if any, e.g., by proving the knowledge of a password, e.g., using the VERIFY command (described in section 3.7), by proving the knowledge of a key, by secure messaging.

Three security statuses are considered: Global-, file- and command-specific security status.

2.5.2 Security attributes

The security attributes, when they exist, define the allowed actions and the procedures to be performed to complete such actions.

Security attributes may be associated with each file and fix the security conditions that shall be satisfied to allow operations on the file. The security attributes of a file depends on its category (DF or EF), optional parameters in its file control information and/or in that of its parent file(s).

2.5.3 Security mechanisms

The following security mechanisms are defined:

- Entity authentication with password: The card compares data received from the outside world with secret internal data.
- Entity authentication with key: The entity to be authenticated has to prove the knowledge of the relevant key in an authentication procedure.
- Data authentication: Using internal data, either secret or public, the card checks redundant data received from the outside world. Alternately, using secret internal data, the card computes a data element (cryptographic checksum or digital signature) and inserts it in the data sent to the outside world.
- Data encipherment: Using secret internal data, the card deciphers a cryptogram received in a data field. Alternately, using internal data, either secret or public, the card computes cryptogram and inserts it in a data field, possibly together with other data.

2.6 APDU message structure

An operating system handles all the communications between the software and the hardware. The SCOS is no different, but the ways in which operating systems communicates varies. A step in application protocols consists of sending a command, processing it in the receiving entity and sending back the response. Therefore a specific response corresponds to a specific command, referred to as command-response pair. An application protocol data unit (APDU) contains either a command message or a response message, sent from the interface device to the card conversely.

A smart card OS uses APDU commands to communicate with the hardware, most of the commands used to communicate are defined in the standard. Every command falls into a certain class of commands. All commands defined in the standard belong to the class of “cross industry commands”. In addition to this class of commands most OS also have other application specific classes of commands. This section will discuss this message structure and also specify the different commands implemented in SCOS.

2.6.1 Command APDU

As illustrated in the figure 2.3, the APDU consists of a mandatory header of four bytes (CLA INS P1 P2) and a conditional body of variable length.

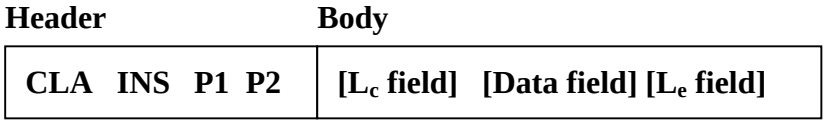

Figure 2.3 Command APDU structure[11].

The number of bytes present in the data field of the command APDU is denoted by L_c and the maximum number of bytes expected in the data field of the response APDU is denoted by L_e.

2.6.2 Response APDU

Illustrated by figure 2.4 the response APDU defined in this part of ISO/IEC 7816-4 consists of a conditional body of variable length and a mandatory trailer of 2 status bytes (SW1 SW2).

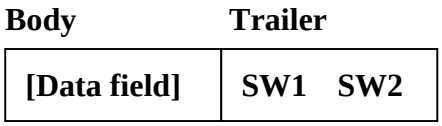

Figure 2.4 Response APDU structure[11].

2.7 Coding conventions

Because of the fact that this chapter is just meant to summaries what is stated in the ISO/IEC 7816-4 standard,[11], and give a glimpse of the important theory behind our work, we will not dig deep into the coding conventions of the different APDU commands. Instead we will focus on describing the specific coding conventions concerning the commands we have implemented. We will do this later in this chapter. But the following notations (table 2.1) are mandatory to comprehend before looking at the coding specifications of the commands.

CLA	One byte that describes the class of the instruction.
INS	One byte that contains the instruction code.
P1	One byte that holds the first instruction parameter.
P2	One byte that holds the second instruction parameter.
L_c	The length varies between 1 and 3 bytes. Holds the data length.
Data field	The length is specified by L_c. Contains a string of data.
L_e	Length varies between 0 and 3. Maximum bytes expected in the response.

Table 2.1 Command APDU structure[11].

2.8 Basic inter-industry commands

In this section we will describe those commands that are specified in the standard and implemented in the SCOS application. For information about the other basic commands we refer to the ISO/IEC 7816-4 standard specification document[11].

The **READ BINARY** command gives a part of an EF as response message. The command is processed on the currently selected EF. Security status must satisfy the security attributes defined for the current EF.

The command message look like in table 2.2.

CLA	Class of the command.
INS	'B0'.
P1	Short identifier (read other EF than current).
P2	Offset (from where to read).
L_c field and Data field	Empty.
L_e	Number of bytes to be read.

Table 2.2 Read Binary command message[11].

The response message looks like below:

Data field	Data read (L_e bytes).
SW1-SW2	Status bytes.

Table 2.3 Response APDU structure[11].

The **SELECT FILE** sets a current file, on success. Selecting a DF (may be MF) sets it as current DF. Selecting an EF sets a pair of current files: the EF and its parent file. After the answer to reset, the MF is implicitly selected.

The SELECT FILE command message looks as follows:

CLA	Class of command
INS	'A4'
P1	Selection control
P2	Selection options
L_c field	Empty or length of the subsequent data field
Data field	If present, according to P1 and P2, file identifier path from the MF path from current DF DF name
L_e field	Empty or maximum length of data expected in response.

Table 2.4 Select file command message[11].

The **VERIFY** command initiates the comparison in the card of the verification data sent from the interface device with the reference data stored in the card (e.g., password). The security status may be modified as a result of a comparison. Unsuccessful comparisons may be recorded in the card.

The command message has the following structure:

CLA	Class of the command.
INS	'20'
P1	'00'
P2	Qualifier of the reference data.
L_c field	Empty or length of the subsequent data field.
Data field	Empty or verification data.
L_e field	Empty.

Table 2.5 Verify command message[11].

The **GET RESPONSE** command is used to transmit from the card to the interface device APDU(s) (or parts of APDU's) which otherwise could not be transmitted by the available protocols.

The command message structure looks like below:

CLA	Class of the command.
INS	'C0'
P1	'00'
P2	'00'
L_c field	Empty.
Data field	Empty.
L_e field	Maximum length of data expected in response.

Table 2.6 Get response command message[11].

2.8.1 Command sequence

To gain a greater understanding of the APDU commands an example of a command sequence could be useful. Let us for instance say that we want to read the data of a binary file called DataFile located in DirectoryFile, which parent file is the MasterFile, located at the root. First, you have to make DataFile your current EF. To do that you have to select the file, using the SELECT command. In the beginning of this command sequence example, your current DF pointer points to the root. Therefore, your first action will be to select DirectoryFile.

2.8.2 Selecting and reading (example)

To select a file the SELECT command is used. The SELECT command is written in hexadecimals and will have the following appearance: “A0 A4 01 00 00”. The Hexadecimals “A0” stands for the class - cross industry commands. “A4” stands for the command SELECT and “01” tells the operating system that it is a DF that should be selected. Following this SELECT command is another SELECT.

Now with the purpose of selecting the DataFile, making that file the file to which the current EF pointer points to. The SELECT command will in that case have the following appearance: A0 A4 00 00 00.

When the file to be read have been selected the only thing left to do is to issue the READ BINARY command, with the following appearance: A0 B0 00 00 08. “B0” stands for the READ BINARY command and “08” tells how many bytes that should be read. When the last command is finished processing you will get the data in the binary file as a response.

3 The development environment

One of the essential characteristics of integrated circuits for Smart Cards is the high level of system security. They have only one input/output through which the interior of the chip is neither visible nor controllable. This makes application code difficult to design and test. Atmel has developed an emulation tool to facilitate the development of an AT90SC application[3]. We will just briefly describe the tool and look at the most essential parts. Let us begin with a look at a basic sketch of the development tools (fig 3.1).

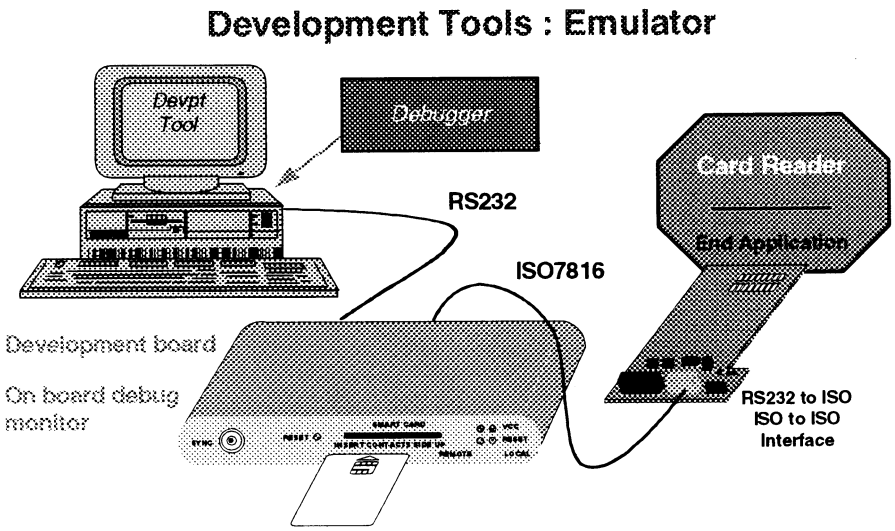

Figure 3.1 Development tools [3].

3.1 AT90SC development kit

The AT90SC Development Kit (SDK) is hardware equipment designed to assist the design engineer in the development of software. The version of the AT90SC device that the SDK is built around has an additional bus connection used to connect an external RAM that emulates the internal memory (Flash/EEPROM) of the production version.

The joint use of the SDK and the AT90SC development chip provides application designers with means of defining an application in terms of the ISO 7816 standard and testing the code using devices that monitor its step-by-step execution.

The SDK has an embedded ISO interface used to communicate either with the development chip, or with the embedded smart card reader. The smart card reader allows the programming of an application on-chip.

3.1.1 Block diagram of SDK

The SDK is a system structured around an AT89C52 micro controller. It is made up of a motherboard, a daughterboard and an extension board (fig 3.2).

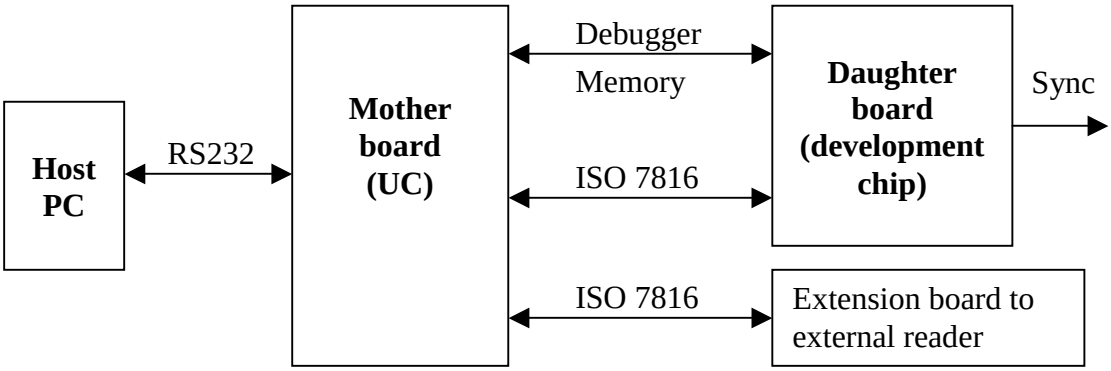

Figure 3.2 SDK block diagram[3].

3.1.2 Motherboard

Inside the motherboard you will find:

- a micro controller (AT 89C52) connected to a PC by an RS232 interface
- an SRAM for tools (SDK software) execution
- an EEPROM for tools memorization
- an ISO interface connected with the development chip or the local reader
- peripherals for system configuration.

The **SRAM** is used to download the tools delivered with the SDK and the application code of the smart card to be uploaded via the ISO interface. The size of the SRAM is 128K bytes, of which only 64K bytes are used. The remaining bytes are reserved for future use.

The **EEPROM** allows the different software tools delivered with the SDK to be saved and makes the user interface easier. Its size is 32K bytes. The **ISO interface** permits communication with the development or production chip in order to download and program an application. The **RS232 interface** is used to connect the SDK with a PC, using the COM port. This is mandatory for the use of the SDK.

3.1.3 Daughter board

The daughter board contains the following:

- an SRAM memory emulation
- an SRAM memory for hardware breakpoints feature
- a DPRAM for debugger data transfer
- line drivers and level shifts for ISO – CMOS conversion
- the AT90SC development chip.

The **DPRAM** (16K bytes in size) is used to transfer messages between the debugger on the PC (in our case C-SPY, described later) and the ROM monitor (executed by the development chip). The **SRAM memory emulation** replaces the internal memory (Flash/EEPROM) of the AT90SC chip. The **breakpoint memory** is, not surprisingly, used to implement the hardware breakpoints. When the user sets a breakpoint it is memorized in this memory at the location corresponding to the address where the breakpoint is set. When the address is fetched, a hardware signal is sent to the development chip, as an interrupt, and stops program execution.

3.1.4 Extension board

The extension board provides the interface between the development chip on the SDK and an external reader. It is composed of:

- an ISO footprint contact for reader connection
- an HE10 with ISO signals for SDK connection
- a DB9 for RS232 connection with external equipment.
- an ISO to RS232 converter
- a jack connector to supply power.

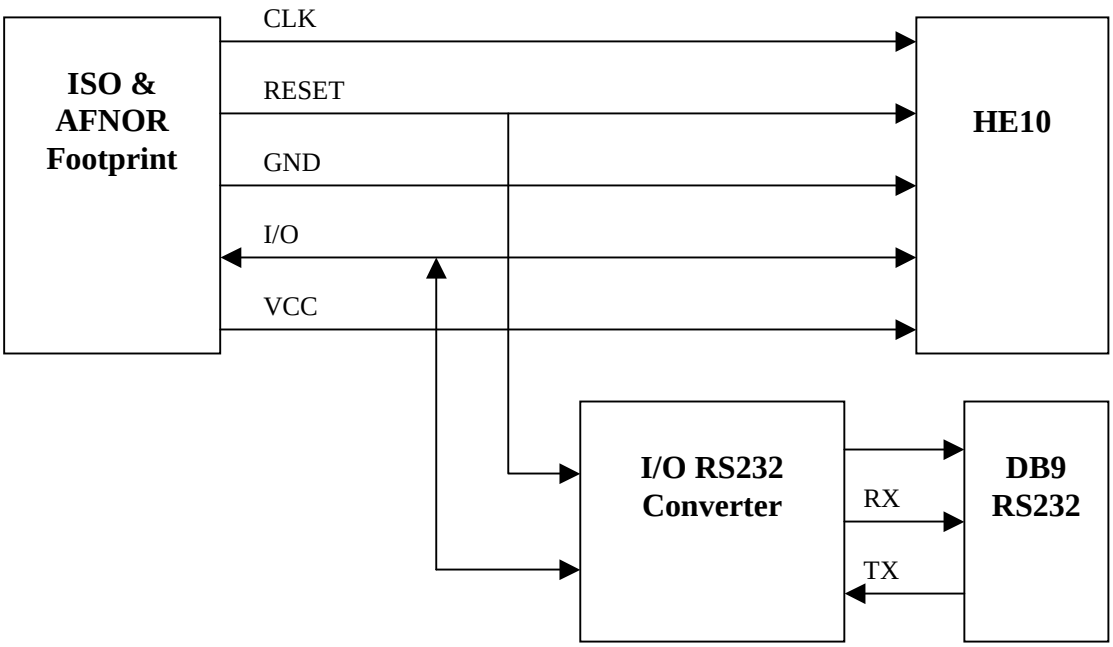

Figure 3.3 Block diagram of Extension board[3].

3.2 AT90S IAR C compiler

The IAR Systems AT90S C compiler, [5], is available in two versions: a command line version, and a Windows version integrated with the IAR Systems Embedded Workbench development environment[7]. In our work we used the latter mentioned version of these two compilers.

IAR Systems C Compiler for the AT90S family of microprocessors provides all the standard features of the C language[5]. In addition many extensions have been added to allow the user to take advantage of the AT90S-specific facilities. AT90S C Compiler provides some features (according to the Programming guide, but not all features are confirmed), which we will mention here[5].

The compiler has conformance to the ANSI specification and it also has a standard library of functions applicable to embedded systems, with source optionally available. It has fast compilation and a memory-based design that avoids temporary files or overlays. For a complete list of the features, we refer to the appendix.

Despite all the features mentioned here and in the appendix, there are some things that are not supported. The AT90SC IAR C Compiler lack the support for templates, multiple inheritance, exception handling, run-time type information, new cast syntax and name spaces.

3.3 AT90S IAR C-SPY

C-SPY for windows, that we used, is a high-level language debugger for embedded applications. The C-SPY debugger allows you to switch between C level and assembler level debugging[6].

During C level debugging you can execute the program one C statement at a time, and monitor the values of C variables and data structures.

Assembler level debugging executes the program one assembler instruction at a time, and displays the registers and memory or changes their contents.

Some of the features that C-SPY offers are an intuitive Windows interface, C and a assembler level debugging and a fast simulator. For a complete list of the features, we refer to the appendix.

3.4 AT90SC micro controller

To fully understand the capacity of the microprocessor that we worked with you had to read the entire AT90SC datasheet[8], but for the purpose of this paper it is enough to give you a short glimpse at the microprocessor and its features (see Figure 3.4). The AT90SCC family is based on the AVR enhanced RISC architecture. The fast-access register file contains 32 x 8-bit general purpose working registers with a single clock cycle access time.

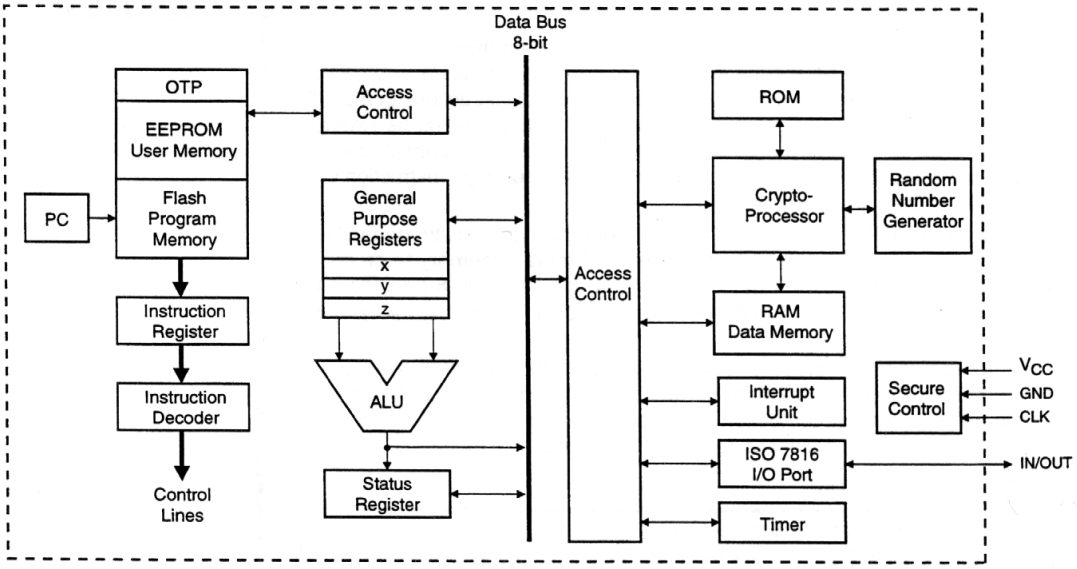

Figure 3.4 The AT90SC Series AVR Enhanced RISC Architecture [8].

3.4.1 ALU

The ALU (Arithmetic Logic Unit) operates in direct connection with each of the 32 general-purpose working registers. One instruction can therefore be executed in a single clock cycle, and by doing that the AT90SCC achieves peak throughputs of close to one MIPS per MHz. After execution the result is stored in the register file in one clock cycle. The ALU operations are divided into three main categories: arithmetic, logical and bit-functions. Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing, enabling efficient address calculations.

3.4.2 Fast encryption

The crypto engine featured in the AT90SCC series is a 16-bit processor dedicated to perform fast encryption or authentication functions. Additional security features include power and frequency protection logic, logical scrambling on Program Data and addresses, and memory accesses controlled by a supervisor mode.

The frequency protection logic only allows communication with the chip within a certain frequency span. If the communication is not within this span no contact can be made with the chip.

3.4.3 Device	**Program Memory Bytes**	**User Memory EEPROM Bytes**	**3.4.4 RAM Bytes**	**Crypto-processor**
AT90SC1616C	16K Flash	16K	1K	Yes
AT90SC3232	32K Flash	32K	1K	No
3.4.5 AT90SC3232C	**32K Flash**	**32K**	**1K**	**Yes**

Table 3.1 The AT90SCC Family[9].

In the appendix we have a list of features provided by the AT90SC.

4 Work analysis

This chapter is the core of our report. Here we will define exactly what we implemented when developing the SCOS, how that was done and also we will try to motivate our design decisions made during the development. However, first we will specify the parts already developed by Atmel. These parts were the cornerstones in our operating system. In this chapter, we presume that you have read the theory chapters.

4.1 Method

Because of our lack of knowledge in the problem domain we decided to tackle the problem in an explorative manner. The first thing we did before starting our explorative programming process was discussing how our file structure should be designed. This was probably the most critical stage in our project. A misjudgement here and we would have encountered huge problems later on and be forced to start all over again. Of course before designing the file structure we tried to get familiar with the development environment. One stage in that process was our visit to the Atmel factory in Rousset, France.

When we felt that we were sufficiently familiar with the environment and the process of development, we were ready to start the explorative implementation process, i.e. we began implementing a small functional part, then we simulated and debugged that part, and after that we started implementing another small part (see Figure 4.1). This process was repeated until the project was finished.

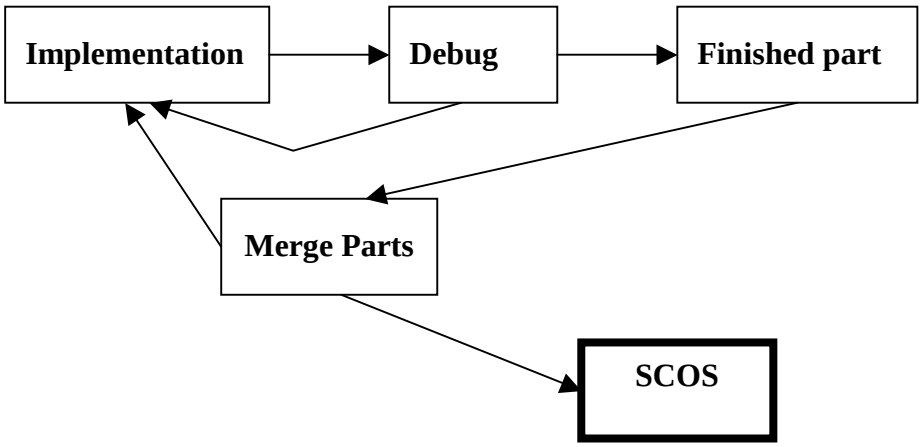

Figure 4.1 Simple sketch of our development process.

4.2 Foundation

Together with the development environment, Atmel provided us with certain foundation parts, essential for us in order to be able to succeed with our work in this very short timeframe. These parts mainly deals with the hardware, like memory management, I/O management and other hardware functionalities. We will very briefly mention these parts because it is important to know that they exist. To make it easier to explain what parts already were developed we have chosen to divide the different parts in three foundation stones, memory management, I/O management and hardware definitions.

4.2.1 Memory management

Routines for addressing the memory, restoring the memory, writing to the memory and transporting data between RAM and EEPROM are provided with this foundation. These parts were very important to us particularly when we developed the file system in the SCOS.

4.2.2 I/O management

In order to communicate with the card we required low level I/O management, i.e. hardware close I/O routines. Atmel provided us with this management, i.e. means to pass information on the data busses to our more high-level defined variables. The hardware close I/O management was exclusively written in Assembler so we had not time to analyse the code and therefore we could not come up with more specific information about this part.

4.2.3 Hardware definitions

As the header suggest, this foundation stone consist of all the definitions used in the SCOS. Of course, these definitions are not only used by our implementation but also the memory management and I/O management rely upon these definitions. In addition to these definitions we have defined are own definitions. We will discuss them later in this chapter.

4.3 File structure

The file structure was by far the most extensive part of developing an operating system. In this section we will describe the design of the file structure and discuss why we implemented it in the chosen way.

4.3.1 Files

Two types of files exist in SCOS, elementary files and dedicated files. We will from now on refer to them as EF and DF. In addition, we have the Master file (referred to as MF), which is a DF but with one slight difference, it is 4 bytes shorter. This is due to the fact that it does not need a father pointer or a brother pointer because of the simple reason that it could not have a father or a brother (see Figure 4.2). The MF is always the root of the file structure tree. Notice the difference in structure compared to what we described in the theory chapter. We choose to implement a father pointer, even though it was not a prerequisite according to the international standard. We thought that we needed this pointer to fulfil the requirements stated in the standard.

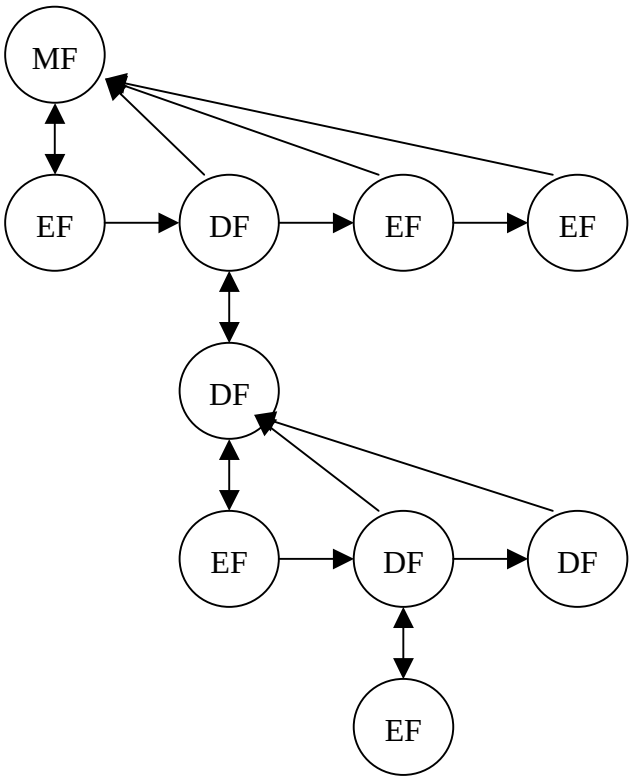

Figure 4.2 Logical file organization in SCOS (example).

4.3.2 EF file structure

The EF structure is always transparent (binary). Records are not supported in SCOS. The reason for this is our wish of doing a simple, but a none the less fully functional, operating system. We prioritized functionality in our project instead of complexity, i.e. we focused on making SCOS executable with possibility to expand rather than aiming far too high in the beginning ending up with nothing to present, nothing working that is.

In figure 4.3 the complete EF header structure is shown. In addition there is the binary data, also called the body, which length is stated at the last byte of the header.

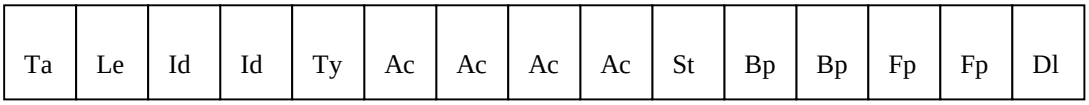

Figure 4.3 EF's header structure, the abbreviations are described in table 5.1.

Ta (1 byte)	Tag that indicates in what format the header is structured.
Le (1 byte)	States the length of the header.
Id (2 bytes)	File identifier, used for file reference.
Ty (1 byte)	File type, EF or DF.
Ac (4 bytes)	Access conditions, purpose described later.
St (1 byte)	Status byte, purpose described later.
Bp (2 bytes)	Brother pointer, described in detail later.
Fp (2 bytes)	Father pointer, described in detail later.
Sp (2 bytes)	Son pointer, not displayed above because of reasons described later.
Dl (1 byte)	States the length of the body (the data).

Table 4.1 Abbreviations.

4.3.3 DF file structure

DF contains no data, just file information and file references to the DF's first child. The header of the DF is identical with the header of the EF but with two distinct differences, that the DF's son pointer is in use (see Figure 4.4). Space for a son pointer in the EF header does exist, but is always empty and can never be allocated a pointer, therefore you could call those bytes to be reserved for future use (RFU). The other difference is that DF´s data length field never will specify any data length. So the last header byte of DF is also RFU.

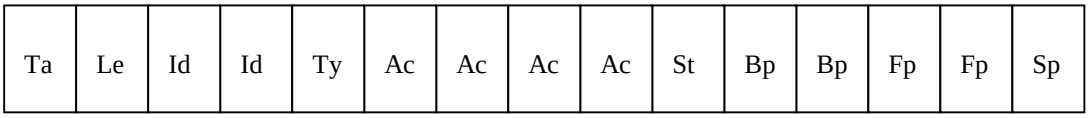

Figure 4.4 DF's header structure, abbreviations described on next page.

Ta (1 byte)	Tag that indicates in what format the header is structured.
Le (1 byte)	States the length of the header.
Id (2 bytes)	File identifier, used for file reference.
Ty (1 byte)	File type, EF or DF.
Ac (4 bytes)	Access conditions, purpose described later.
St (1 byte)	Status byte, purpose described later.
Bp (2 bytes)	Brother pointer, described in detail later.
Fp (2 bytes)	Father pointer, described in detail later.
Sp (2 bytes)	Son pointer, described in detail later.
Dl (1 byte)	Data length, not displayed in figure because of reasons described later.

Table 4.2 Abbreviations.

4.3.4 MF file structure

The structure of MF is exactly the same as the structure of the DF. Even though the fact that MF does not have a Brother pointer or a Father pointer, we chose to implement these bytes as RFU, because we wanted to be able to use same offset parameters, for all files, when referring to different spots in a files header. This was a choice we did early in the development process. We thought that it would make it easier for us having a uniform format for the header structure, in fact that is still our opinion.

4.4 Header definitions

The structure of the header files was implemented according to the suggestions in the standard. But we had to make some choices on our own. Below we will try to describe and motivate the choices we made.

4.4.1 Tag and length

The Ta byte and the Le byte are both fundamental parts that define the structure of the header. The Ta byte is a tag that specifies the format of the header, while Le describes the length of the header. All our headers are the same length and use the same format, but that is only due to the constraints we introduced before starting developing SCOS. We have only two different file types and we chose to implement these file headers in the same way to preserve simplicity.

4.4.2 File identifier

To be able to reference one distinct file in our file system we need a way to separate them from each other. In order to achieve this we introduced a file identifier (the file identifier was suggested by the standard, but not required). We will describe how we chose to implement file referencing in the next chapter.

4.4.3 File type

Not surprisingly this byte indicates the type of the file. This might seem as overkill since our file system only have two different file types, but again when we implemented SCOS our intention was to develop a basic operating system with possibilities to expand.

4.4.4 Access conditions

We dedicated four bytes for access conditions and to be honest it was maybe one byte too much. But due to lack of knowledge about access conditions at the given time we reserved space for four access condition bytes. However, the fourth byte does not necessarily need to be used for that specific purpose. It could easily be changed and used for anything else. Therefore we like to call the last byte for RFU (reserved for future use).

The first nibble, starting from left in the access conditions bytes of the header, is used to indicate whether file creation/reading is allowed or not. In the EF header the nibble indicates reading access, while in the DF header the nibble indicates the file creation conditions for the files below that specific DF.

The second nibble, in the DF case, stands for conditions for deletion and in the EF case for the possibilities to update.

The third and fourth nibble respectively, contains the access conditions for rehabilitate and invalidate. We do not actually use these two conditions, but a short explanation seems to be in order. It is in fact quite simple, when you invalidate a file it is put out of use, like it does not exist. However, the file can be rehabilitated, which means that it can be put in to use again. We have the necessary foundation for this function to exist, like access conditions and state indicator, but we did not have the time to implement state checks and treatment of the different states. The remaining four nibbles, like we said earlier, are not in use at this time. However, for the first two of these four nibbles there are suggestions of use in the standard.

The purpose of having a nibble to describe whether something is allowed or not does not seem very effective. For this purpose one bit should be enough, but this is a result of us preparing for future expansion. It is common to have different levels of security. For example if the nibble is set to 1 you need verify PIN-code number 1 and if the nibble is set to 2 you need to verify PIN-code number 2 and so on. This sort of security levels is used in SIM-cards developed for cell-phone users. In GSM 11.11 you can read more about security levels[12].

4.4.5 Status byte

The purpose of the status byte, at least in our minds, is to indicate whether or not the files is active, that is if the file is invalidated the status byte is set to 1 and if it is valid it is set to 0, or vice versa. Like we said when we discussed access conditions for invalidate and rehabilitate, this is not fully implemented, and due to that fact not in use.

4.4.6 Pointers

We have implemented three different types of pointers, a son pointer, a brother pointer and a father pointer (see Figure 4.2). The root in the structure (MF) has of course only one pointer, the son pointer. Space for brother and father pointer is despite this fact implemented. We chose to do that that way because we wanted the headers to be of the same size, so we could use the same offset parameters for all headers.

The son pointer of a specific dedicated file points only to the first (oldest) son of that specific dedicated file. To be more precise, the pointer is an address reference to the spot in memory where the sons file header begins. The last statement applies to all three types of pointers.

The brother pointer of a specific file, DF or EF, points to the specific files closest brother to the right in the file structure, that is, if a file B is the first son of a file A,

then B´s closest brother C is A´s second son. A does not have a direct connection to C, but with A´s connection to B, and B´s connection to C via the brother pointer, A has an indirect connection to C.

Finally we have the father pointer. Every file, DF or EF, which has a father, that is, every file except the root file (MF), has a father pointer. The father pointer points, not surprisingly, to the specific files father. We knew when we designed the file system that we needed a way to move “backwards” in our file structure and therefore we chose to implement a father pointer. Looking back, a double linked brother pointer probably would have been a better choice. But with our solution we satisfied the demands stated in the standard, which was our purpose. We will dig deeper into pointers in the next two sections when we discuss file referencing and search method.

4.5 File referencing

In order to manage your files you have to be able to reference them. There were a couple of possible ways of doing that suggested in the standard. Below we will describe the ways of file referencing we chose to implement.

4.5.1 EF referencing

To be able to reference an EF you have to know the file identifier of that file. Also, the current DF must be that EF´s father. In conclusion, if a file under the current DF matches the file identifier defined in the input, that file will be referenced.

Of course the above explanation of EF referencing does not contain the entire truth, first you need to find exactly the place in memory where the different file identifiers resides so that you can identify it by comparing that specific identifier with the file identifier got on the terminal input.

In order to search among the file headers in our file system, trying to find the matching file identifier, we use pointers. This search method solution that we use in SCOS will be described in detail in the next section, now let us focus solely on the file referencing.

4.5.2 DF referencing

There are two ways of referring DF´s implemented in SCOS. Either you refer to the current files father or you refer to the oldest DF-child of the current DF. This means that you can move “up” and “down” in the file structure, which was our intention. With the possibility to move up and down as described, in conjunction with the possibility to refer to a distinct file on the current level, we have the means to select any file we want to select in our file structure. In figure 4.5 you can se an example of file referencing.

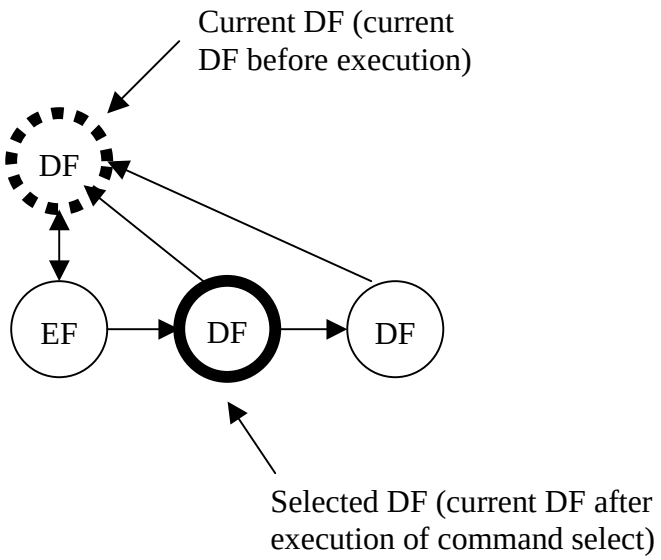

Figure 4.5 Example of DF child selection.

4.6 Security

The security in our operating system is limited, due to a low priority in the development process. The security in SCOS partly resides on PIN codes. We have implemented two PIN codes with full functionality. With full functionality we mean that different security levels (different pin codes, in our case two) is supported, a verification of the PIN code is done and we also have the possibility to decide the value of the PIN code.

We focused on making it possible to verify the PIN code, the management of the codes is unfortunately not to our satisfaction. The PIN code resides as plain text in the memory and with the right knowledge it is possible, but not likely, for a person to read the codes.

However, the security mainly resides in SCOS as an operating system. The only actions that can be done by a user is through the APDU-commands that we have implemented. This makes unauthorized actions towards the card difficult.

Other than the security measures mentioned above we have not implemented anything to assure security. Of course there are some security related to the card and the implementations done by Atmel, but that we will not discuss in this report. Mainly because of confidentiality reasons and also because lack of knowledge.

4.7 Implemented functions

With the purpose to give some kind of knowledge of what it is exactly that we have done we will specify the functions we have implemented in our operating system. We will not discuss them on a programming code level, but we will explain the purpose of the different functions. The functions are:

- **Create MF**

In this case the name it self explains what the function is used for. It is a proprietary APDU (OS specific, not standardized) command that creates the MF. In our OS a MF has to be created before any other commands can be successfully executed. In the function we mainly implement the header of the file, and also certain controls are made, that is checking if the right security level is satisfied etc.

- **Create File**

The Create File function is also called by a proprietary APDU command. The actions of this function are quite similar to Create MF. The difference is that this command should be able to create two different types of files, DF and EF. Like in Create MF some controls are made and pointers are updated.

- **Update File Structure**

Whenever a file is created the file structure tree have to be updated. The Update File Structure function does all the necessary controls in order to update the pointers in all files needed in a correct way, except the pointers in the file that was created. The pointers in the created file are updated by the Create File function. This command could not be issued by an APDU command, only by the Create File function.

- **Select File**

The Select File function is issued by a standardized APDU command with the same name. This function has means implemented for searching in the file structure, compare file identifiers and updating current pointers.

- **Verify**

The Verify function, like select file, also implements the actions of a standardized APDU command. It handles the comparison between an input value and a stored PIN code.

- **Read Binary**

The Read Binary standardized APDU command reads the current file (last selected EF) and returns the data to the terminal. Of course, security level check according to the access conditions for reading that specific file is executed.

- **Update Binary**

Update Binary is also issued by a standardized APDU command. The function writes a body to a EF, that is, it fills a created file with data. Security checks are made and the files header variables are updated.

- **Check Stats**

Stats is set to positive when a succeeded verify command has been executed. The Check Stats function manage the security checks, that is, it checks the current security level and compare that to the security level necessary for executing a specific command. If the current security level satisfies the needed security level a positive response is returned to the calling function.

- **Clear Eeprom**

The Clear Eeprom function was implemented with the sole purpose of making the implementation and test process easier for us. We could during development issue this function with a simple command from the terminal.

4.8 Testing

The development environment made it pretty easy to test our OS. After each compilation we could load the code to the SDK:s SRAM and then simulate a card using the extension board described in the theory chapter. We placed the extension board in a standalone card reader and used a simple APDU application to send APDU:s to the simulated card.

We could not to full extent test the separate commands until all commands was fully implemented. Let us say that we first implemented the UPDATE BINARY command. We could execute this command and receive an “OK” from the card saying that the APDU was functional, but we had no idea if the correct data was written to the file before we could read it. So we had to implement the READ BINARY command and use that command to se if the UPDATE BINARY command really worked correctly.

In the final test we loaded SCOS to a real smart card and issued all of our commands with success. We managed to create a tree of files, update the EF:s binary part, verify PIN code etc.

5 SCOS put into practise

In this chapter we will present an example of a possible “real world” usage of SCOS. We think that this will help to provide greater understanding of what SCOS really is. There might be some abbreviations not explained in detail in this chapter, for further knowledge we refer to chapter 2.

5.1 History

A cell phone is every man property these days, at least in the industrial world. All modern cell phones has to have a certain card installed, otherwise there will be no connection with the network. This card is called a SIM, Subscriber Identity Module. The SIM provides the phone with an identity, which needs to be authenticated by the service provider that issued the SIM-card. If no successful authentication has taken place, there will be no connection to the network.

This was the sole purpose of the early SIM-cards. Today, however the SIM-cards have broaden its field of use. We will not describe every feature found in the SIM-cards of today, but one particular feature we will describe in detail, with the purpose of anchoring the use of SCOS in a “real world” context.

5.2 The phonebook

Every SIM-card has a phonebook where you can store all your contacts. The good thing about storing phone numbers on the card is that you can access your phonebook from any cell phone, as long as you insert your SIM-card in that phone, of course. Everyone that has a cell phone probably sometime has stored a number on his SIM-card. But what happens inside the phone and the card when a user stores a number in the phonebook?

5.3 ADN

First and foremost, the SIM-card have to have a file called ADN (Abbreviated Dialling Numbers). This file is created using administrative commands (create file) during the personalisation process of the card. These administrative commands have been described earlier in this report.
A personalisation process is in short words, the creation of the card. During this creation process the card gets an identity, a phone number, a phonebook etc. The phone book is empty after this creation process, except for possibly some customer support numbers provided by the service provider.

5.4 Selecting a stored number

Let us describe the whole course of events when Sam wants to have Marks number displayed, Marks number that is stored in Sam’s phonebook. In the user context the only thing that happens is that Sam selects Mark in his phonebook. But what happens on a lower level? How does the phone retrieve Marks number and display it on the screen? Well here it is.

The phone sends five consecutive APDU commands in the following order, SELECT MF, SELECT DF_TELECOM, SELECT EF_ADN, READ_RECORD No (Mark), and finally GET_RESPONSE. The first command selects the root in the file structure of the SIM-card. The second command selects a certain DF usually called TELECOM. Then the phonebook is selected with the command SELECT EF_ADN. The specific structure of those commands is described in the theory chapter. Finally the phone issues the READ_RECORD command. When this command reaches the SIM-card, the card stores the contents of that specific record, containing information about Mark, in a buffer. This data is in reach by the phone using the command GET_RESPONSE. When that command is issued, the data, in this case Marks phone number is sent to the phone. At this point Sam can see Marks phone number on his cell phones display. All the commands used for this action are implemented in SCOS.

6 Conclusion

In this chapter we will discuss what we have accomplished, what parts could have been done differently, what the next step is and what we have learned. That is, we will analyse the results and see if they meet our primary goals.

6.1 Results

Let us begin with examining the functional results of our work, that is, finding out if the final product meets our goals and expectations. Everything that we thought were essential to meet the expectations we wrote down in our requirements specification, we have implemented these parts. But was that enough?

Also, let us conclude if we managed to answer the questions raised in the problem discussion.

6.1.1 Functionality

When testing and analysing SCOS in its current state we can establish the fact that we meet our functional goals. We have an operating system that consists of a simple file system were we can refer to different memory locations using our file structure. We have implemented the following commands: Read binary, Update binary, Create file, Create MF, Verify and Select file. With these commands implemented we can say that we have developed a basic operating system that is fully functional.

6.1.2 Answered questions

We have discovered that implementing a basic operating system does not need to be an expensive project, neither referring to time or money. Of course the quality of the final product is strongly related to the amount of money and time at your disposal.

During the development the importance of the file system became very clear to us. We consider the file system as the fundamental part of our operating system.

Looking back to the beginning of our project we can see how important knowledge in all areas of the problem domain really is. In order to make a meaningful operating system you first need to know which kind of functionality that it has to provide. When you have determined the functions to implement you have to analyse what kind of foundation you need to have to be able to provide that specific functionality.
For example, in our case, we very soon discovered that in order for us to make a meaningful OS we had to implement a select file function. At that time we understood that we also needed to design a file system, otherwise the select file function could not been put in operation. It is quite obvious that if we like to select a distinct file among a whole bunch of files we need a way to search among them. That could never be done without a file system.

The need of cryptography is, even though we did not implement it in SCOS, very clear to us. For example, storing PIN-codes in plain text as we do in SCOS is quite silly. An intruder could easily retrieve and read a certain code on the card if that code is not encrypted. But of course the importance of cryptography depends in what field the card will be put into operation and how security sensitive that field is.

6.2 Alternative solutions

In this section we will try to find out what parts of our work that could have been done in other ways. Of course there is an immense selection of choices that could have been made differently during the development, but let us focus on the parts that we, with the knowledge we have today, know for certain could have been done better.

6.2.1 File system

We knew, already during the development, that the design of the file system was not the optimal one. However, the shortage of time forced us to implement the first design that crossed our minds and then move on. We will not present any alternative solutions on how to design a better file system, only just come to the conclusion that it with great possibility could have been done better. For example, our file structure is not fully dynamic in the sense that we could not increase the length of data in an EF.

6.2.2 Security

Although we have implemented PIN and administrative keys we have to admit that our operating system is not even close to secure. We only implemented the different keys, which are unprotected, to be able to confirm that our verify command was functional. As mentioned earlier in this chapter, cryptography is one part that could have been implemented to increase the security of our operating system.

6.3 Future of SCOS

The next step in the SCOS project is to decide whether to begin from scratch or if it would be justifiable to expand the operating system from its current state. It is indeed possible to expand the existing product, because during development we were careful to make SCOS expandable. However, with the knowledge we have today we know that some parts of our work, despite our efforts of making it expandable, limits the possibilities of future expansion. For example it is not possible to increase the size of a data file after its creation. To make that possible requires a bit of work.

The conclusion is that if we only want to do a minor increase in functionality, SCOS is fully to our satisfaction. However if we have a commercial interest, we can without any doubt say that we need to do major changes in our implemented modules. There are of course parts like the interpreter and some ADPU-commands that we could use when developing a new operating system, but most of the file structure has to be redone.

However, the most important thing to remember if we should do more or less a new operating system is to be even more careful in our choices of design.

6.4 Gained knowledge

In this part we will summarize some of what we have learned during this project. The first thing that we realized, were that an operating system for a smart card wasn't what we expected. We soon understood that the image we had of an operating system for smart cards were quite different in reality. Maybe it was a bit naive but we compared the application you interact with on your cell phone with the operating system in a computer, for example Windows. In our way to look at it we were supposed to develop a user interface rather than an operating system. The difference is without exaggerating very big. So during this project, you could say that we learned what an operating system really is.

Another thing that we have learned is the way in which smart cards communicates with the outside world. Also we have gained a lot of knowledge about memory management at a low level, that is, how the memory stores data, how a flash memory works, how you refer to different places in the memory, etc. As we mentioned before, we have come to the conclusion that the most important part of the operating system is the file system and its structure.

7 Definitions and abbreviations

7.1 Definitions

Command response pair: Set of two messages a command followed by a response.
Data unit: The smallest set of bits that can be unambiguously referenced.
Dedicated file: File containing file control information and optionally, memory available for allocation. It may be the parents of EF's and/or DF's.
Elementary file: Set of data units or records which share the same file identifier. It cannot be the parent of another file.
File control parameters: Logical, structural and security attributes of a file.
File identifier: A 2-bytes binary value used to address a file.
Internal elementary file: Elementary file for storing data interpreted by the card.
Master file: The mandatory unique dedicated file representing the root of the file structure.
Message: String of bytes transmitted by the interface device to the card or vice-versa.
Parent file: The dedicated file immediately preceding a given file within the hierarchy.
Password: Data that may be required by the application to be presented to the card by its user.
Record: String of bytes that can be handled as a whole by the card and referenced by a record number or by a record identifier.
Record identifier: Value associated with a record that can be used to reference that record. Several records may have the same identifier within an elementary file.
Record number: Sequential number assigned to each record that uniquely identifies the record within its elementary file.
Working elementary file: Elementary file for storing data not interpreted by the card.

7.2 Abbreviations

ALU	Arithmetic logic unit
APDU	Application protocol data unit
ATR	Answer to reset
CLA	Class byte
DIR	Directory
DF	Dedicated file
EF	Elementary file
INS	Instruction byte
MF	Master file
P1-P2	Parameter bytes
PKI	Public key infrastructure
PTS	Protocol type selection
RFU	Reserved for future use
SDK	Smartcard development kit
SW1-SW2	Status bytes
TLV	Tag, length, value

8 References

Books:

[1] Baltimore Tech. An introduction to public key infrastructure - **pki**, Baltimore Technologies, 1999

[2] Bruce Schneier **Applied cryptography** - second edition, Wiley, 1996

Manuals:

[3] Atmel **AT90SC Smart Card Development Kit** - users manual, Atmel, 1998

[4] Atmel **SDK Manager -** users manual, Atmel, version 1.3

[5] IAR Systems **AT90S IAR C compiler** – prog. guide, IAR Systems, 1996

[6] IAR Systems **AT90S IAR C-SPY** - users guide, IAR Systems, 1996

[7] IAR Systems **AT90S IAR Embedded Workbench** - interface guide, IAR Systems, 1996

Datasheets:

[8] Atmel **AT90SC Addressing Modes & Instruction Set**, Technical data, Atmel Smart Card ICs, 2001

[9] Atmel **AT90SC3232C** - Atmel Registered Confidential Proprietary, Technical data, Atmel Smart Card ICs, 2000

[10] Atmel **AT90SC Bootloader** - Atmel Registered Confidential Proprietary, Application Note, Atmel Smart Card ICs, 2000

Standards:

[11] ISO/IEC **ISO/IEC 7816-4** – International Standard, ISO/IEC, 1995

[12] ETSI **GSM 11.11** – technical specification, ETSI, 1999

APPENDIX A

CRYPTO

This section regarding crypto was meant to be a part of the theory chapter, but when we decided that we will not be able to implement crypto as it was stated in then problem statement we moved this section from the theory chapter to this appendix. The section will introduce to you the basics of cryptography.

Cryptography

Data that can be read and understood without any major difficulties is called **plaintext**. If you **encrypt** a plaintext, the result will be a text that is incomprehensible, called **ciphertext**. The process of making the ciphertext readable again is called **decryption.** The art of cryptography is the art of using mathematics to encrypt and decrypt data. Crypto is accordingly about securing data while crypto analysis is about analysing and cracking a secure communication.

Secret keys

Cryptography secures the confidentiality by encrypting a message with a secret key in conjunction with an algorithm. This procedure creates a not readable version of the message that the recipient later can decrypt, by using the original key, and retrieve the readable contents. The key that has been used has to be kept a secret between the two parties and that is the main difficulty with the most crypto applications, i.e., to handle and keeping these keys a secret.

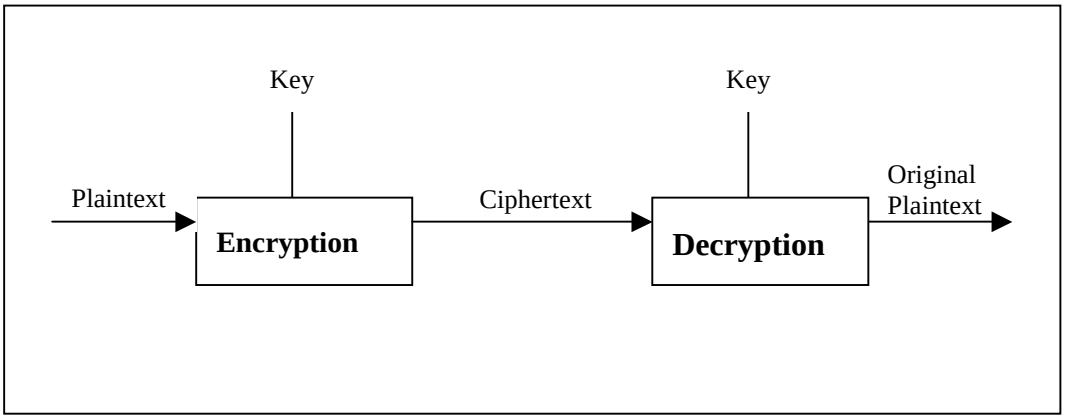

Figure A.1 Encryption and decryption with a key.

Public key infrastructure
The action of securing business and communication over computer networks can be seen as an electronic equivalence of signing letters and have it in at glued envelope. The signature proves that it is authentic and the glued envelope makes sure that it stays confidential.

Public key cryptography
The problem with ordinary cryptography is solved when replacing the secret key with a pair of keys, one private and one public. Information that is encrypted with a public key can only be decrypted with its private counterpart. As an addition to encryption, the public and private key can be used to create and verify digital signatures. These can be added to the messages to make the message and sender clear.
PKI[1] provides the main frame for a great variation of components, applications and the means for combining and reaching the four most important security functions for commercial transactions:

- Confidentiality – to keep the information private.
- Integrity – to assure that the information has not been manipulated.
- Authentication – to assure the identity of an individual or application.
- Non-repudiation – to assure that the information not will be deprived.

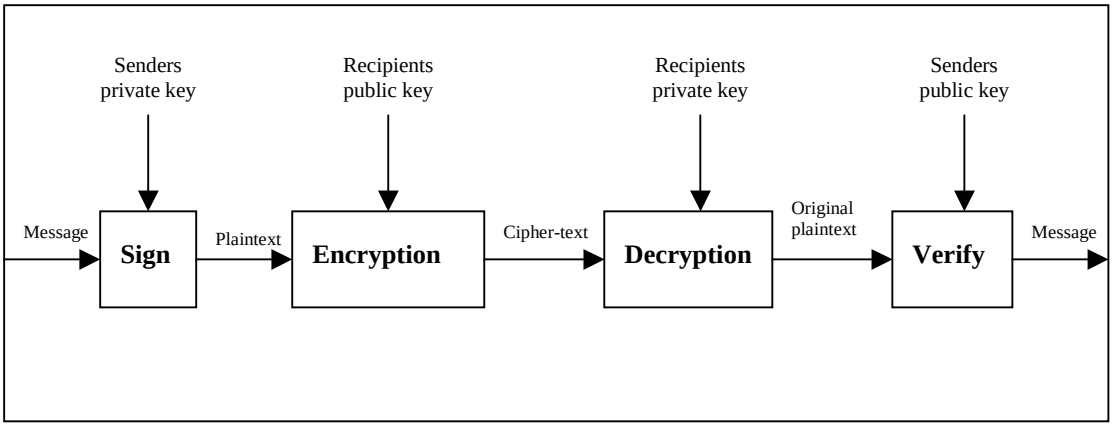

Figure A.2 The sender signs his message with his private key (encrypts for example his name at the end of the message). After that the sender encrypts the entire message with the recipients public key. The recipient decrypts the cipher text with his private key and then verifies the sender by decrypting the signature with the senders public key. Then the recipient knows that the message is in fact sent from the sender and that nobody else has been able to read the message.

PKI components
A PKI is a combination of hard- and software-products and procedures. These are the parts that a PKI should consist of:

- A Security Policy.
- Certificate Authority (CA).
- Registration Authority (RA).
- Certificate Distribution System.
- PKI-enabled Applications.

Security policy
The policy contains, among other things, statements about how the organisation should handle keys and other valuable information. The policy also sets the control level that is necessary to match the level of risk.

Certificate authority
The CA system is the trust basis of a PKI as it manages public key certificates for their whole life cycle.

The CA should issue a certificate by binding the identity of a user or a system to a public key with a digital signature. Schedule due day for certificates. And also assure that the certificate cease to exist when necessary by publishing Certificate Revocation Lists (CRL).

Registration authority
The RA provides the interface between the user and the CA. It captures and confirms the identity of the users and hand over the certificate inquiry to the CA.
Certificate distribution system
Certificates can be distributed in a number of different ways depending on the structure in the PKI surroundings, e.g., of the users themselves or through register service.

Examples on PKI applications:

- Communication between web servers and browsers.
- E-mail.
- Electronic Data Interchange (EDI).
- Credit card transactions over the Internet.
- Virtual Private Networks (VPN).

RSA
RSA[2] is a ”public key” algorithm, easy to understand and to implement. Given a name after it’s three creators, Ron Rivest, Adi Shamir and Leonard Adleman. RSA rely it’s security on the difficulty of the factorisation of big numbers. The public and private keys are functions of a pair of big primes.

Key generation
To generate the two keys, you choose randomly two big primes, **p** and **q**. For maximum security the two primes should have the same length.

$$\mathbf{n} = \mathbf{pq}$$

You then choose randomly an encryption key, e, so that e and $(\mathbf{p} - 1)(\mathbf{q} - 1)$ is relatively prime, i.e., **e** and $(\mathbf{p} - 1)(\mathbf{q} - 1)$ have only the number one as mutual divider. You then use Euklides extended algorithm to generate the encryption key, **d**, that:

$$\mathbf{d} = e^{-1} \bmod ((\mathbf{p} - 1)(\mathbf{q} - 1))$$

Note that also **d** and **n** are relatively prime. The numbers **e** and **n** is the public key and the number **d** is the private key. The primes **p** and **q** is no longer necessary, they should be removed and never be revealed.

Encryption
To encrypt a message we divide it in numeric blocks, smaller than **n**. That is if the number **n** is 100 characters long, every message block should be just below 100 characters long. The encrypted message is built on similar blocks and of approximately the same length. The encryption formula looks as follows:

$\mathbf{c} = \mathbf{m}^{\mathbf{e}} \bmod \mathbf{n}$, where **m** is one message block and **c** is one encrypted message block.

To decrypt a message, take each encrypted block and calculate:

$\mathbf{m} = \mathbf{c}^{\mathbf{d}} \bmod \mathbf{n}$, where **c** is one encrypted message block and **m** is one message block.

RSA performance
RSA will never reach the same speeds as symmetric algorithms. But a clever choice of **e** can however make a great difference regarding the speed. The three most common choices of **e** are 3, 17 and 65537.

RSA security
The security of RSA depends solely on the problem with the factorisation of big numbers. It has however never been confirmed mathematically that you need a factor **n** to calculate **m** from **c** and **e.**

APPENDIX B

AT90S IAR C compiler, [5].

LANGUAGE FACILITIES

- Conformance to the ANSI specification.
- Standard library of functions applicable to embedded systems, with source optionally available.
- IEEE-compatible floating-point arithmetic.
- Powerful extensions for AT90S-specific features, including efficient I/O.
- LINT-like checking of program source.
- Linkage of user code with assembly routines.
- Long identifiers – up to 255 significant characters.
- Up to 32000 external symbols.

PERFORMANCE

- Fast compilation.
- Memory-based design that avoids temporary files or overlays.
- Rigorous type checking at time of compiling.
- Rigorous module interface type checking at link time.

CODE GENERATION

- Selectable optimization for code speed or size.
- Comprehensive output options, including relocatable binary, ASM, ASM + C, XREF, etc.
- Easy-to-understand error and warning messages.
- Compatibility with the C-SPY high-level debugger.

TARGET SUPPORT

- Tiny and small memory models.
- Flexible variable allocation.
- Interrupt functions requiring no assembly language.
- A #pragma directive to maintain portability while using processor specific extensions.

APPENDIX C

AT90S IAR C-SPY, [6].
C-SPY offers the following features (according to the Users guide, not all of which are confirmed):

GENERAL FEATURES

- Intuitive Windows interface.
- C and assembler level debugging.
- Fast simulator.
- Log file option.
- Powerful macro language.
- Complex breakpoints.
- Interruptable at all times.
- Memory validation.

C LEVEL DEBUGGING

- C expression analyser.
- Full type recognition of variables.
- Function trace.
- C call stack with parameters.
- Watchpoints on expressions.

ASSEMBLER LEVEL DEBUGGING

- Full support for auto and register variables.
- Built-in assembler/disassembler.
- Optional getchar/putchar emulation.

APPENDIX D

AT90SC, [8].

The list of features provided by the AT90SC (according to the AT90SC datasheet) follows.

- High-performance, Low-power AVR Enhanced RISC Architecture
  - Harvard Architecture
  - All Registers directly connected to the ALU
  - Two Registers can be accessed by a Single Instruction in a Single Clock Cycle
  - 120 Powerful Instructions
  - Most Single Clock Cycle Execution
- Up to 32K bytes Flash Program Memory
  - Endurance: 100 000 Write/Erase Cycles
- 1K bytes RAM
- Crypto-coprocessor
  - Pre-programmed Functions for Cryptography and Authentication
- Supervisor Mode (Memory Management)
- ISO 7816 I/O Port
- Random Number Generator
- 16-bit Timer
- 2-level, 5-vector Interrupt Controller
- Security Features
  - Power-down Protection
  - Low-frequency Protection
  - High-frequency Filter
  - Logical Scrambling on Program Code
- Low-power Idle and Power-Down Modes
- Bond Pad Locations Conform to ISO 7816
- 2.7V to 5.5V Operating Range

Växjö universitet

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

tel 0470-70 80 00, fax 0470-840 04
www.msi.vxu.se