MASARYK UNIVERSITY, FACULTY OF INFORMATICS
UNIVERSITAT POLITÈCNICA DE VALÈNCIA, ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA INFORMÀTICA

Cryptographic functions in a smart card

Final Degree Project

Antonio Bustos Rodríguez

2009/2010, Brno

This Project consists from setting up development environment for a particular class of smart cards and developing cryptographic application that will demonstrate smart card capabilities.

Contents

Introduction

Smart card

A smart card is a card with embedded integrated circuits which can process data. This implies that it can receive input data which is processed and delivered as an output. There are two types of smart cards: memory cards and microprocessor card. Memory cards contain only memory storage components for example to store some data in the card. Microprocessor cards contain memory components and microprocessor components. The smart card is made of plastic, generally PVC, like the older credit cards with magnetic strip and its size is similar to the older credit cards.

There are two technologies to use the smart cards: contact smart card and contactless smart card. Contact smart cards transmit the data with the metal conductors or pins of the card. Contactless smart card transmits the data through radio frequencies and for example, they are used in electronic passports. The smart cards help to make easier the routine actions like user identification, user authentication, data storage, because they are devices with a small size and they can be carried in a simple wallet. An example of usage of a smart card is the user authentication in a large companies or instutions to sign-on or access in restricted areas.

To work with the contact smart card, the card has to be inserted in the smart card reader and when the communication is done between the reader and the contact pads of the card, so it is no necessary add a battery in the smart card because the energy is supplied by the reader to the card. The smart cards are defined in the standards The ISO/IEC 7816 and ISO/IEC 7810. The standards define the physical characteristic of the card, the communication protocol used, basic functionality…

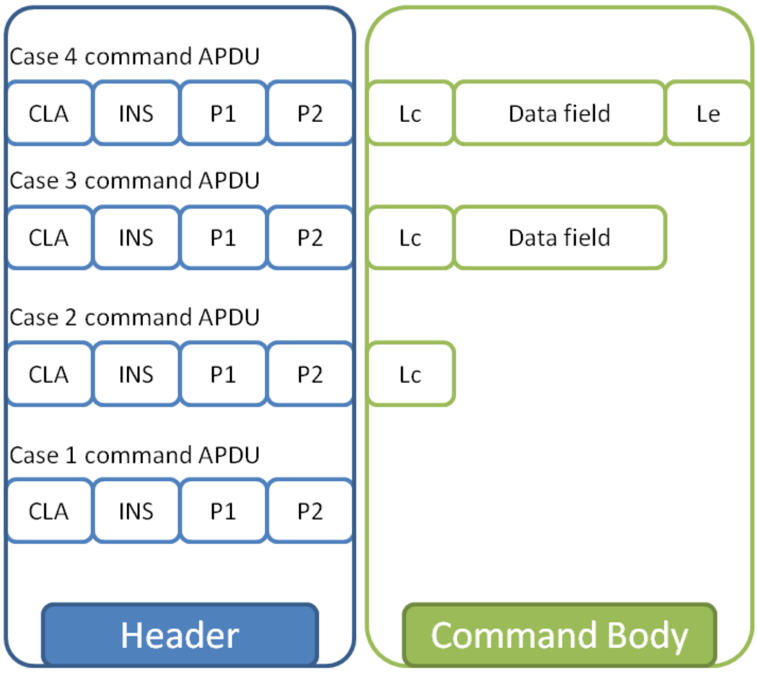
Command APDU

The message structure used to transmit data between the smart card and the smart card reader is the Application Protocol Data Unit, or commonly called, APDU. The structure of an APDU is defined by the ISO/IEC 7816 standards and the concept of the command APDU is similar to the TCP/IP protocol in networks.

A command APDU is composed with a header and with a command body. The header consists of four bytes: Class (CLA), Instruction (INS), Parameter 1 (P1) and Parameter 2 (P2). The class byte indicates the standard which the command is executed, the instruction byte defines the command to execute and the two parameters (P1 and P2) are used to transmit extra information to the command.

The command body is composed with a maximum of three elements: Lc (length command), Le (length expected) and the data field. The length command defines the length of the data in the data field of the command APDU and the length expected contains the length of the data requested from the smart card, which is returned in the data field of the response APDU. The data field is the part where the data is sent to the smart card and it has a maximum length of 256 Bytes.

A command APDU has four different combinations, and in the next image each combination is called a case. The case 1 does not contain any command body, the case 2 contains the byte “Length command” from the data field, the case 3 is similar than the case 2 but in this case there are transmitted in the data field and the case 4 is the full command APDU.



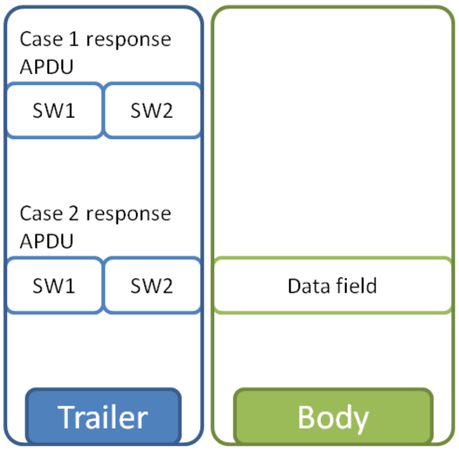
Structure of the command APDU

More information in the chapter “6.5.1 Structure of the command APDU” from Smart Card Handbook (2004), Wolfgang Rankl and Wolfgang Effing.

Response APDU

The response APDU is the reply of a command APDU and it is composed of the status word and the data field. The status word (SW1 and SW2) informs about the processing status of the command execution and it is composed of two bytes and the standard defines a serie of status word. The data filed contains the data returned from the smart card wich was processed into the smart card. The length of the data field can be specified in the byte Le of the previous command APDU and the data field has a maximum length of 256 Bytes.

There are two variants for a response APDU and they are related in the next image. The first variant is a reponse APDU without data field and the second variant returns data to the reader.



Structure of the response APDU

More information in the chapter "6.5.2 Structure of the response APDU" from Smart Card Handbook (2004), Wolfgang Rankl and Wolfgang Effing.

Transmission protocols

There are two transmission protocols to transmit the data between the smart card and the reader. The T=0 transmission protocol transmit the data per bytes and has a poor layer separation.

The T = 1 transmission protocol transmit the data per blocks and is an asynchronous half-duplex protocol for smart cards. It is an improvement of the previous protocol, because it has a strict layer separation. This protocol features strict layer separation, which means that data destined for higher layers, such as the application layer, can be processed completely transparently by the data link layer.

A block in the T = 1 tranmission protocol consists of a prologue field, an information field and a epilogue field. The only part that is optional to send is the information field, which is composed by the command or response APDU. The structure of a block is showed below.

prologue field			information field	epilogue field
node address (NAD)	protocol control byte (PCB)	length (LEN)	APDU	EDC
1 byte	1 byte	1 byte	0 ... 254 bytes	1 ... 2 bytes

The structure of a T = 1 transmission block

The prologue field consists in three bytes: node address (NAD), protocol byte (PCB) and length (LEN). The node addres contains the destination and source addresses for the block, the protocol control byte supervises the transmission protocol and the length indicates the length of the information field, in other words, the command or response APDU.

The information field contains the command APDU and his content is not analyzed or used in this layer. The epilogue field is used to detect errors in the transmission of the block.

More information about the T = 1 transmission protocol in the chapter 6.4.3 (p. 409) from Smart Card Handbook (2004), Wolfgang Rankl and Wolfgang Effing.

Communication with smart cards

PC/SC (Personal Computer/Smart Card) is specification to integrate smart cards in a computing environment, particulary Windows environments, needing a driver of the smart card reader compatible with this specification. PC/SC is implemented in almost all the Microsoft operating systems like Microsoft Windows 2000, XP. For other computing environments, like Mac OS X or Linux there is a free implementation called PC/SC lite.

This specification allows to the applications to work directly with the smart card and it is available in the most used programming languages like C, C++, Java and Basic.

MUSCLE (Movement for the Use of Smart Cards in a Linux Environment) is a project created to coordinate the development of drivers for smart card readers and a API to help the applications to communicate with the smart cards in a Linux environment.

At first, it was started because for a long time the drivers required for using smart card with Linux were not available, but they were appearing, it was necessary establish an interface in Linux to work with smart card, for example, for operations such as logging on.

With regard to its architecture, MUSCLE is strongly passed on PC/SC, but in contrast to PC/SC the source code is available under a GPL license.

Java Card Technology

Java Card is a technology to develop applications to run in Java compatible smart cards. This applications, or commonly called applets, are designed to run securely on smart cards, because they are executed with a Java Card Virtual Machine inside the smart card. The Java card applets are very portable, because they can be executed in any Java compatible smart card and the applets are executed securely because this technology offers data encapsulation, an applet firewall, cryptographic functions and the features of the Java language.

Java Card 3.0 is a new version of this technology and it has two editions. The Classic Edition is an evolution of the previous version of Java Card and supports the previous java card applets developed. The Connected Edition includes a new virtual machine and a new environment to execute applets with network-oriented features.

However, we cannot use this technology with our smart cards because they are incompatible.

Module scard

The smartcard.scard module is a library to communicate in C language with the smart card readers compatible with the PC/SC specification. The module is the lower layer of the pyscard framework and it is an application programming interface to work with the smart card and provides the next functions:

- SCardAddReaderToGroup
- SCardBeginTransaction
- SCardCancel
- SCardConnect

- SCardControl
- SCardDisconnect
- SCardEndTransaction
- SCardEstablishContext
- SCardForgetCardType
- SCardForgetReader
- SCardForgetReaderGroup
- SCardGetAttrib
- SCardGetCardTypeProviderName
- SCardGetErrorMessage
- SCardGetStatusChange
- SCardIntroduceCardType
- SCardIntroduceReader
- SCardIntroduceReaderGroup
- SCardIsValidContext
- SCardListInterfaces
- SCardListCards
- SCardListReaders
- SCardListReaderGroups
- SCardLocateCards
- SCardReconnect
- SCardReleaseContext
- SCardRemoveReaderFromGroup
- SCardSetAttrib
- SCardStatus
- SCardTransmit

Answer to reset (ATR)

The smart card sends an Answer to Reset (ATR) after the smart card is inserted in the smart card reader, in other words, after the supply voltage, clock signal and reset signal have been applied. The ATR is made up of a data string, which contains various parameters related to the transmission protocol and the characteristics of the smart card. This data string, which contains at most 33 bytes, is always sent with a divider value (clock rate conversion factor) of 372 in compliance with the ISO/IEC 7816-3 standard.

In the main code of the smart card, the ATR is sent at the first to send to the reader the information about the smart card, but this information is provided by the external library included in the code.

More information in the chapter “6.2 Answer to Reset (ATR)” (p. 377) from Smart Card Handbook (2004), Wolfgang Rankl and Wolfgang Effing.

Advanced Encryption Standard (AES)

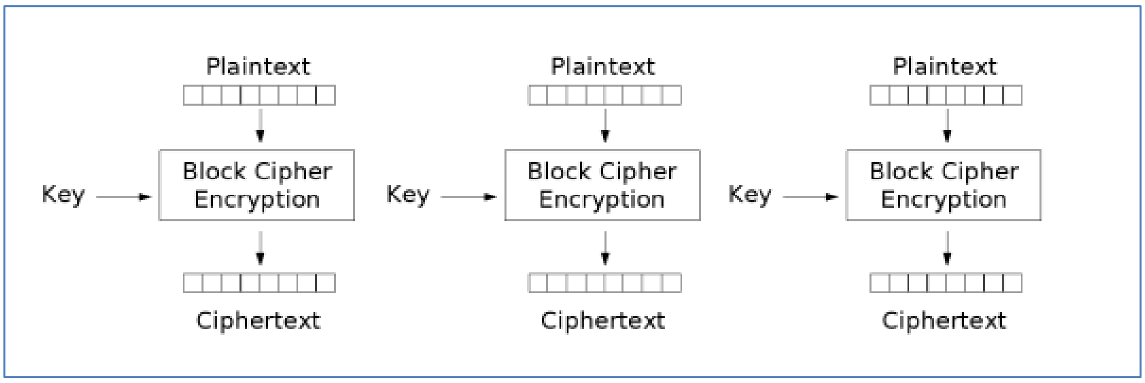
The Advanced Encryption Standard is a symmetric-key encryption standard based in the principle known as a Substitution permutation network. AES has a fixed block size of 128 bits and a key size of 128, 192, or 256 bits, but the block and the key sizes can be any multiple lengths of 32 bits. The blocksize has a maximum of 256 bits, but the key size has not theorically any maximum.

This algorithm came up as improvement of the Data Encryption Standard (DES) and nowdays, it is used deeply and it has been analyzed extensibely and some attacks have been published. This algorithm can be executed in an 8-bit microcontroller, like the microcontroller incorporated in the smart card used and it is used in other environments successfully.

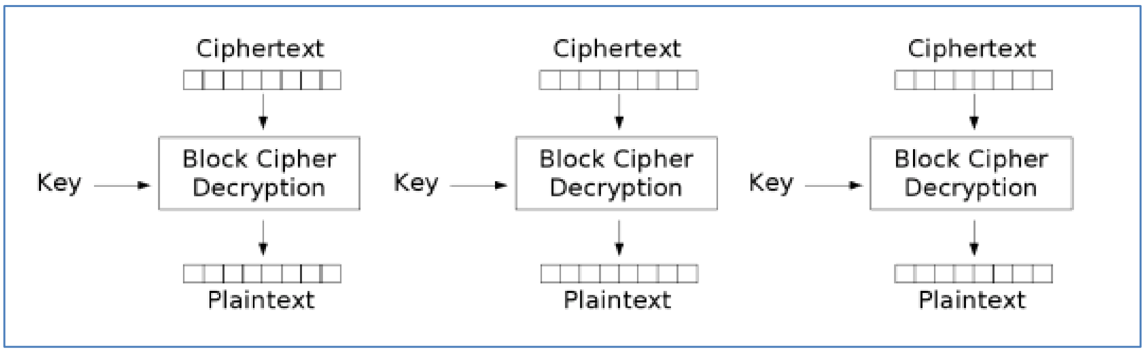
About the security of AES, in the Advanced Encryption Standard article in Wikipedia there is a section about this topic, for more details, consults the Bibliography.

Electronic codebook (ECB)

The electronic codebook is an encryption mode to encrypt data in the symmetric-key algorithms and it is the simplest mode. In the electronic codebook, the message is divided in blocks of the same length and they are encrypted or decrypted sperately, like appears in the below figure.



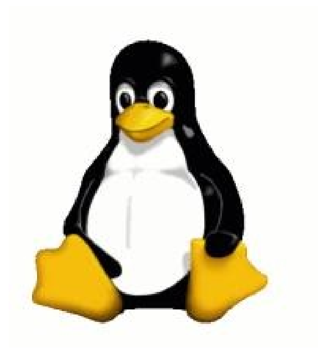
Electronic Codebook (ECB) mode encryption

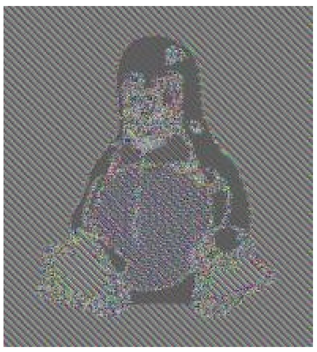


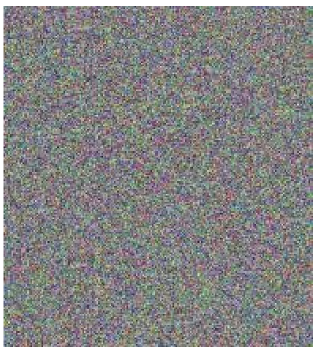
Electronic Codebook (ECB) mode decryption

If the plain text is longer than the block length, it has to be separated in blocks of a determinate length and if the length is not multiple of the block length, in the last block has to be added padding. It is not recommended to encrypt big amounts of blocks with the same key, because this encryption mode always returns the same byte encrypted of a byte of plain text. For example, if the same block to encrypt appears more than once in the plain text, this block is always the same cipher text.

For lengthy messages, the ECB mode may not be secure. If the message is highly structured, it may be possible for a cryptanalyst to exploit these regularities. For example, if it is known that the message always start out with certain predefined fields, then the cryptanalyst may have a number of known plain text-cipher text pairs to work with. If the message has repetitive elements with a period of repetition a multiple of b bits, then these elements van be identified by the analyst. This may help in the analysis or may provide an opportunity for substituting or rearranging blocks. (Cryptography and Network Security: Principles and Practice, William Stallings, 2010, p. 200).


Original


Encrypted using ECB mode


Other modes than ECB results in pseudo-randomness

The first image corresponds to the original message, the next image is the message encrypted using ECB and the last image is the message encrypted with improved versions of ECB. In the original message appears some parts that are identical, like the colours, so these parts always produces the same cipher text and it is easily to identify the original image.

More information in the chapter "6.2 Electronic code book" (p. 198) from Cryptography and Network Security: Principles and Practice (2010), William Stallings.

AVR timers / counters

Timer / Counter 0

Timer0 is an 8 bit timer/counter which can count from 0 to 0xFF in the microcontroller ATmega163. The used registers are:

- Timer registers
    - TCCR0 (Timer/Counter 0 Control Register)
    - TCNT0 (Timer/Counter 0 Value)
- Interrupt registers
    - TIFR (Timer Interrupt Flag Register)
    - TIMSK (Timer Interrupt Mask Register)
    - GIMSK (General Interrupt Mask Register)

In the timer mode of operation, the timer is provided by an internal signal. After each clock cycle the value of the TCNT0 register is increased by one. The clock rate is x times the oscillator frequency. The factor x can have the following values: 1, 8, 64, 256 and 1024 (for example: 1024 - the timer is increased after 1024 cycles of the oscillator signal).

This prescaling is controlled by writing one of the following values into the register.

Initial value	Used frequency
1	ck
2	ck/8
3	ck/64
4	ck/256
5	ck/1024

Timer / Counter 1

In contrast to timer 0, timer 1 is a 16-bit timer/counter in the microcontroller ATmega163.It can be used for longer counting sequences and the counting extent is between 0x0000 and 0xFFFF. The used registers are:

- Timer registers
    - TCCR1A (Timer/Counter Control Register A)
    - TCCR1B (Timer/Counter Control Register B)
    - TCCR1L (Timer/Counter Value Low Byte)
    - TCCR1H (Timer/Counter Value High Byte)
    - OCR1AL (Output Compare Register A Low Byte)
    - OCR1AH (Output Compare Register A High Byte)
    - OCR1BL (Output Compare Register B Low Byte)
    - OCR1BH (Output Compare Register B High Byte)
    - ICR1L (Input Capture Register Low Byte)
    - ICR1H (Input Capture Register High Byte)
- Interrupt registers
    - TIFR (Timer Interrupt Flag Register)
    - TIMSK (Timer Interrupt Mask Register)
    - GIMSK (General Interrupt Mask Register)

In the timer mode of operation, the timer is supplied by an internal signal. After each clock cycle the meter reading is increased by 1. This signal is produced by n times the amount of the oscillator signal. The factor x can have the following result: 1, 8, 64, 256 and 1024 (for instance: 1024- only after 1024 cycles of the oscillators the timer is raised, the frequency is only fosc/1024). These results can be set with register TCCR1B.

The timer is adjusted through writing the following results into the register initial value used frequency.

Initial value	Used frequency

1	ck
2	ck/8
3	ck/64
4	ck/256
5	ck/1024

Objectives and work plan

Motivation

In this period, called by many people information age, the protection of the information is an aim for the society, and more and more people understand it like a right. In the science computer, the protection of the user space is a fascinating objective because it needs improvement constantly to prevent the attacks. The user space is represented with all the files and messages which the users work daily and this user space can be more important than what users think, because more and more, the life of the users is exposed to the computer world.

The only solution to guarantee this security to the users, it is use the cryptographic tools. The cryptography has to guarantee the next properties:

- Privacy. The files and messages used by a user have to be secret to everybody and privacy is sometimes related to anonymity.
- Integrity. The files and messages received to a user have to be exactly in the same than the issuer user sent, and by the way anybody should modify the data transmitted.
- Authentication. The act of establishing or confirming a user as authentic, that is, that claims made by or about the subject are true.
- Irrefutability. The fact that the sender of the information cannot deny that he has sent the information.

The symmetric-key cryptography refers to encryption methods in which the information is encrypted and decrypted with the same key, and this key is shared by the sender and receiver. It is used to guarantee the privacy of the information, because this information can be accessed only by the holders of the key.

This cryptographic method has been in use for thousands of years with the same concept than now, encrypting or decrypting some information with the same key. For example, in Roman times, the emperors used this technique to communicate between the senior. But since these times, the main problem of the symmetric-key cryptography is sharing the key between the sender and receiver, because it is necessary a secure channel to transmit it. Nowadays, the key is transmitted using an asymmetric cryptography algorithm, for example, when the Secure Shell network protocol.

The smart cards are very used in the security world, because some of them are developed with cryptographic tools. Besides, more and more the smart cards are being used in our normal life, for example, in the phones card, transport cards, identification cards, etc. so the development of smart cards applications is increasing and its importance in the science computer.

Main Purpose

The project work would consist from setting up development environment for particular class of cards and developing cryptographic application. We must be able to communicate with the smart card to send some data and expect this data are returned in an encrypted manner. To do this, we have to write an application on the smart card to encrypt data with an AES algorithm, for example.

Besides to offer the encryption and decryption solutions, the hash functions (SHA-1 and SHA-2) digest going to be developed in the smart card. But the implementations found were incompatible with the microcontrollers, because most of them were designed to 32 bit microprocessors. The attempts to convert the implementations to an 8-bit design failed, and the few implementations that can be compiled by the libc module returned wrong digest.

As a result of this situation, the developing of the hash functions stopped and only the classes implemented on Java still working. These classes allow the communication with the hypothetical hash commands of the smart card.

To work with the smart card commands developed in the smart card, an interface to communicate between the smart card and the computer was developed in Java. This interface is capable to select a file, read it, sent it to the smart card in blocks and save the cipher output to a file.

The software development process used in the project is the spiral model, because the activities are not fixed a priori and they are chosen based on risk analysis and the needs emerged in each step. This model has been chosen because the risk at the first to develop the code in the smart card and for the interface with Winscard Smart Card API.

The work plan of the project was at first be used to the smart card topic and look for the information relative of the topic. After that, I set up my laptop to be ready with all the programs necessary to start the project.

The first step was uploading a hex file into the smart card. Afterwards, I started to develop simple commands in the smart card to be used to the code development and the project started seriously with the adding of the AES implementation and the test executions. The development of the project is described in the appendix “Report project per week”.

The materials used in the project are described in the next section.

Materials and method

The smart cards used in the project were FunCard 5 (Atmel AT90S8515 microcontroller) and FunCard ATmega163 (Atmel ATmega163 microcontroller), which are Atmel cards with microcontrollers and programmable memory.

To connect the smart card with the PC, the PC Twin smart card reader was utilized with the USB connection. To program the smart card, the Dynamite +Plus Smartcard Programmer was used.

The software used to the development of the project has been AVR Studio 4, Eclipse and Cas Studio. The interface between the smart card and the PC was developed under Eclipse, the program of the smart card was developed under AVR Studio 4 and to upload the hex file obtained through the AVR Studio, Cas Studio was used.

AT90S8515

The AT90S8515 (FunCard5 card) is a low-power CMOS 8-bit microcontroller based on the AVR RISC architecture. The frequency of the microcontroller approximately is 1 MIPS per MHz, allowing optimizing power consumption versus processing speed.

The features of the AT90S8515 microcontroller are described below:

- 8K bytes of In-System Programmable Flash
- 512 bytes EEPROM
- 512 bytes SRAM
- 32 general-purpose I/O lines
- 32 general-purpose working registers, connected directly to the Arithmetic Logic Unit (ALU).
- Flexible timer/counters with compare modes
- Internal and external interrupts
- A programmable serial UART, programmable Watchdog Timer with internal oscillator, an SPU serial port and two software-selectable power –saving modes.

The device is manufactured using Atmel's high-density nonvolatile memory technology. The On-Chip In-System Programmable Flash allows the program memory to be reprogrammed In-System through an SPI serial interface or by a conventional nonvolatile memory programmer, in the project, Dynamite +Plus Programmer. By combining an enhanced RISC 8-bit CPU with In-System Programmable Flash on a monolithic chip, the Atmel AT90S8515 is a powerful microcontroller that provides a highly flexible and cost-effective solution to many embedded control applications.

More details about AT90S8515 microcontroller in the official document of Atmel, see Bibliography.

ATmega163

The ATmega163 is a low-power CMOS 8-bit microcontroller based on the AVR architecture. The frequency of the microcontroller approximately is 1 MIPS per MHz, allowing optimizing power consumption versus processing speed.

The features of the ATmega163 are described below:

- 16K bytes of In-System Self-Programmable Flash
- 512 bytes EEPROM
- 1024 bytes SRAM
- 32 general purpose I/O lines
- 32 general-purpose working registers, connected directly to the Arithmetic Logic Unit (ALU).
- 3 flexible Timer/Counters with compare modes
- Internal and external interrupts
- A byte oriented Two-wire Serial Interface, an 8-channel, 10-bit ADC, a programmable Watchdog Timer with internal Oscillator, a programmable serial UART, an SPI serial port, and four software selectable power saving modes.

The On-chip ISP Flash can be programmed through an SPI serial interface or a conventional programmer, in the project, Dynamite +Plus Programmer. By combining an 8-bit CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATmega163 is a powerful microcontroller that provides a highly flexible and cost effective solution to many embedded control applications.

More details about ATmega163 microcontroller in the official document of Atmel, see Bibliography.

PC Twin

PC Twin is a smart card reader developed by Gemalto which handles all types of ISO/IEC 7816 compatible smart cards. The main features of PC Twin are:

- A transparent design to show the inserted card.
- USB or Serial connection (simply by cable insertion). In the project case, an USB connection.
- Modular concept with accessories: stand, floppy disk tray, to simplify logistics and inventory. Not used in the project.
- Supports ISO/IEC 7816 Class A, B and C cards (5V, 3V and 1.8V).
- Reads from and writes to all ISO/IEC 7816 microprocessor cards and supports the transmission protocol T=0 and T=1.
- Supports memory cards using "Synchronous Card API". Short circuit detection.

The human interface of the reader consists in a LED with one color (Green). The LED has 2 states: blinking (waiting card insertion), constant on (card reading / writing). The Cable USB reader has as maximum 1.5m long, USB 2.0 type A connector, power supply through USB port, maximum operating current: 100mA and operating voltage [4.4 – 5.5V].

The API to work with the reader is Microsoft PC/SC environment with associated drivers, CT-API and synchronous Card API for support of memory cards.

The operating systems supported are:

- Windows 95OSR2, NT4.00 for PC Twin in serial mode
- Windows 98, 98SE, Me, 2000, XP, Server 2003, x64 editions, Vista 32, 64 bits, Seven, Server 2008R2
- Win CE 4.1, 4.2, 5.0, 6.0 (USB readers)
- Linux Kernel 2.6 and higher
- Mac OS X Panther, Tiger, Leopard 32 editions (USB readers)
- Support for Solaris, XP embedded (USB readers)

The drivers can be downloaded from support.gemalto.com, and the guide installation for each system appears in the same website.

Dynamite +Plus Smartcard Programmer

The Dynamite +Plus is the evolution of the old Dynamite Programmer dated May 2005, developed by Duolabs. The new Dynamite +Plus is smart card programmer with a size similar as a packet of cigarettes. The Dynamite +Plus works with Cas Studio software.

The technical information about the device is described below:

- Full speed USB Device at 46 MHz internal speed.

- No need for external power supply. The energy is transmited by the USB cable connection.
- USB 1.1/2.0 connection.
- Multilanguage software.
- Fully programmable flash firmware for future software updates.

The programmer supports smartcards up directly via USB. The list of OS cards supported is very long and keep on being updated and it appears in the latest version of Cas Interface Studio. The smart cards PrussianCard3/Funcard5 (AT90S8515 + 24C512) and FunCard ATmega163 (ATmega163 + 24C256) are included in the list of PIC and ATMEL AVR supported microcontrollers smartcards, but the list is very long to show and keep on being updated.

Cas Interface Studio

Cas Studio is a software specifically developed by Duolabs to use Dynamite +Plus programmer. Cas Studio can be executed in Windows 98/ME/2000/XP. The software is able to identify the smart card connected to the smart card programmer and to self-adapt to it by enabling/disabling the appropriate options.

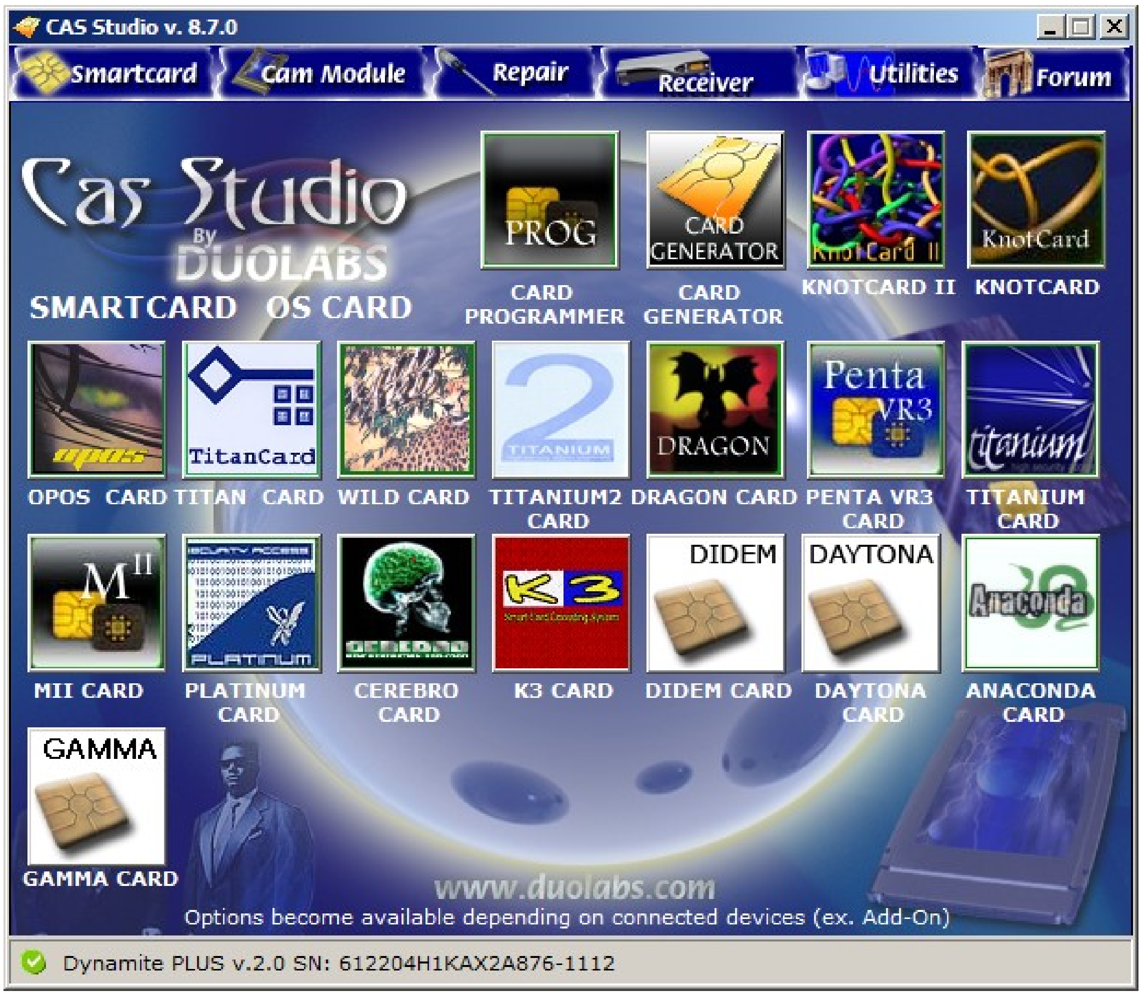
The application needs to be connected to the smart card programmer to start to run, and when the programmer is connected, it display the serial number of the programmer. If a problem occurs or the device cannot be identified, the application displays an error code.

Dynamite PLUS Connected. Waiting to start program ...

The applications is recognizing the device

The top panel of the window displays the menu that enables you to select the category of options you can enable. These are:

- Smart card: It is enabled for the devices Cas Interface 3, Cas Interface 2 + Add-on and Dynamite +Plus. It contains the programming tools for smart cards.
- Cam Module: It is enabled for the devices Cas Interface 3 and Cas Interface 2. It contains the programming options for CAMs.
- Repair: It is enabled for the device Cas Interface 3 only. It contains the reparation options for CAMs.
- Receiver: It is enabled for the devices Cas Interface 3 and Cas Interface 2.
- Utilities: It is enabled for the devices Cas Interface 3, Cas Interface 2 + Add-on and Dynamite.



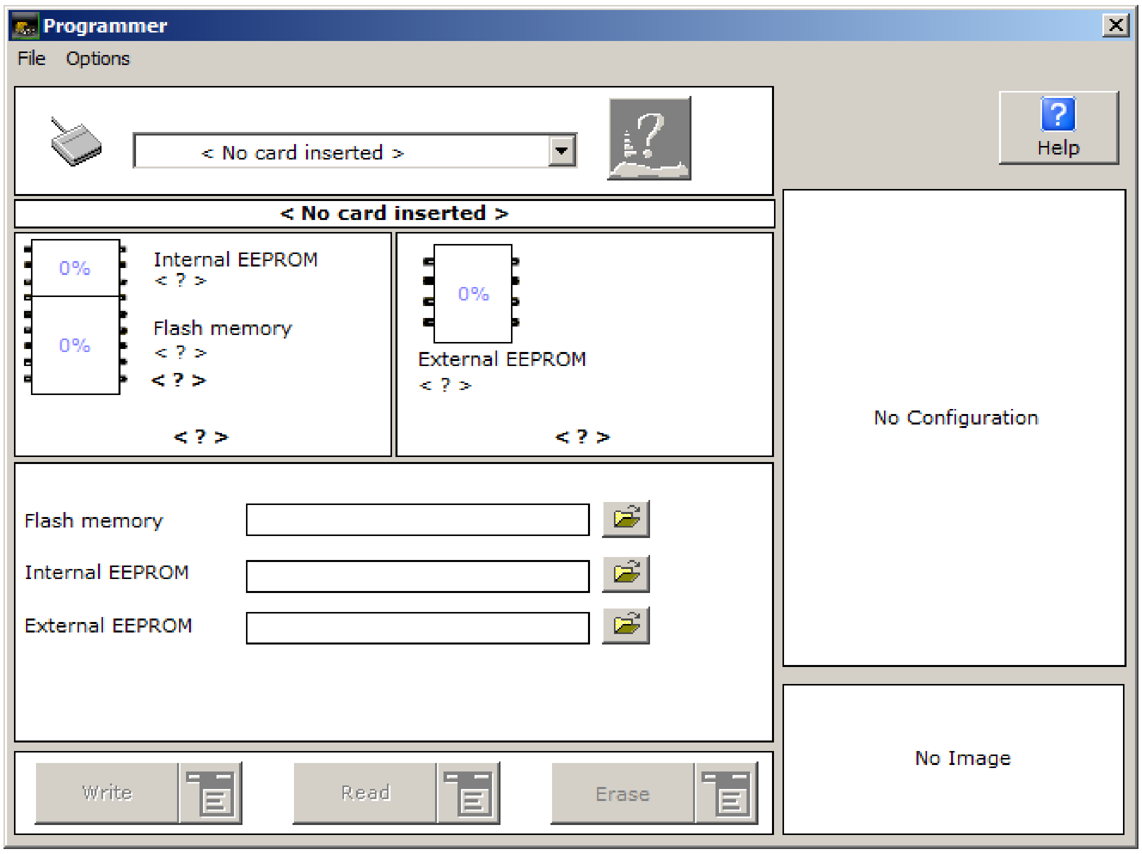
Options become available depending on connected devices (ex. Add-On)

Dynamite PLUS v.2.0 SN: 612204H1KAX2A876-1112

Main screen of the application

To program the PIC-based and AVR-based smart cards supported by Cas Studio, click on the button “Card Programmer”. The following dialog displays below.

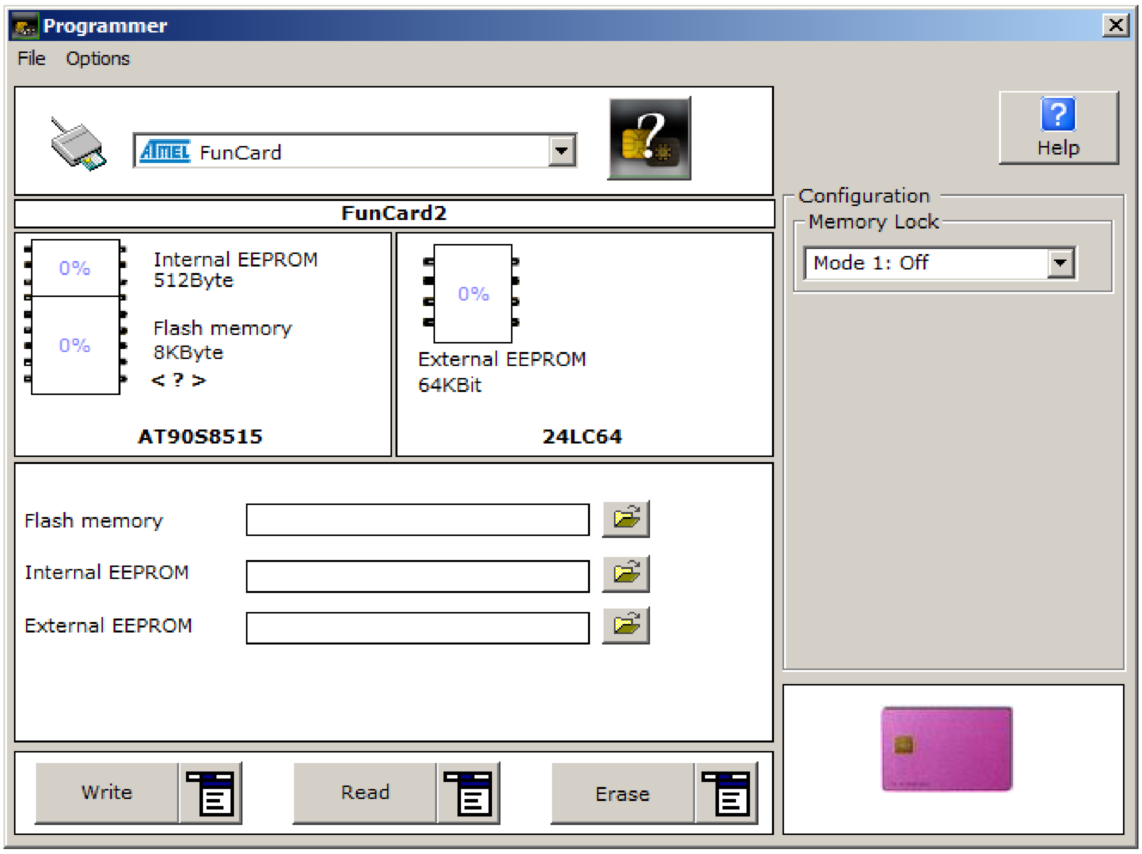
There are available the operations “Read”, “Write” and “Erase” to the smart card. The smart card will be recognized automatically after inserting in the programmer.



Dialog of the Card Programmer

Upload a hex file into the smart card

At first, insert the card into the smart card connector of the Dynamite +Plus programmer and the dialog displayed has to be like the below. The smart card will be recognized automatically, if does not, click the button with the question mark to allow the smart card to be automatically identified.

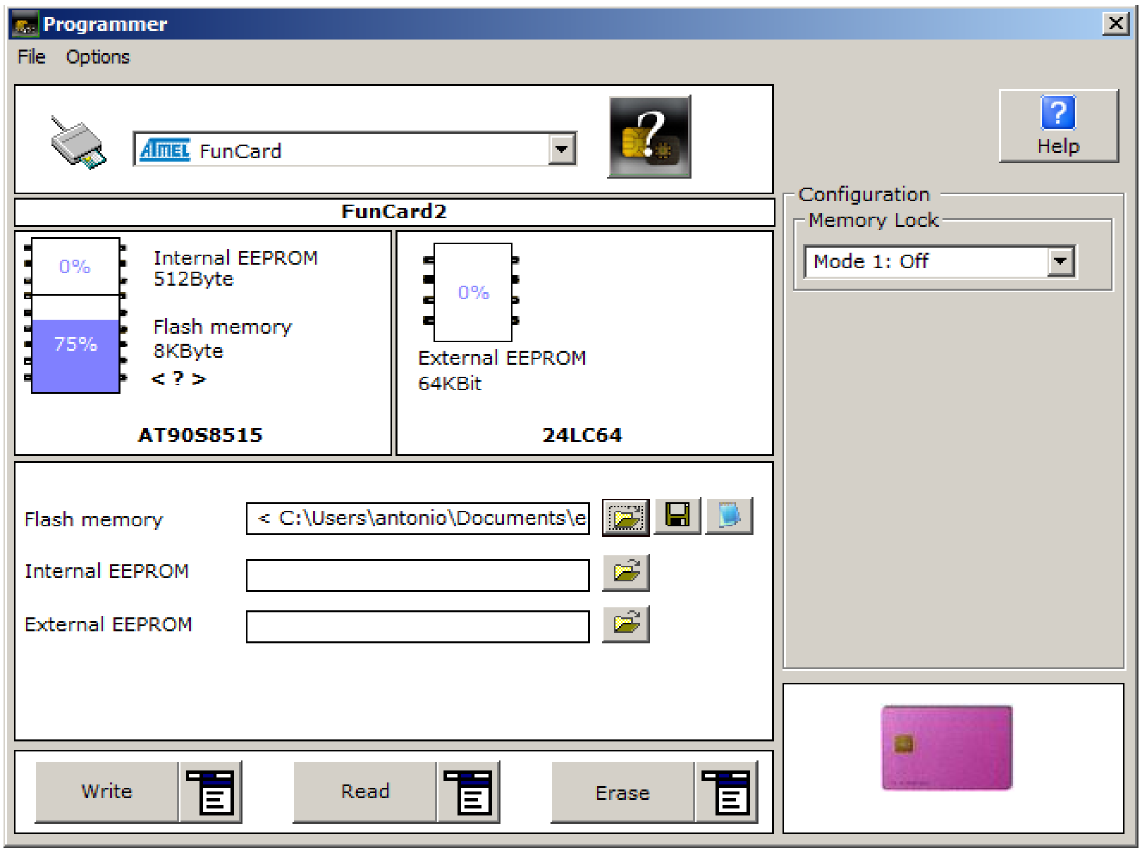


Configuration
Memory Lock
Mode 1: Off

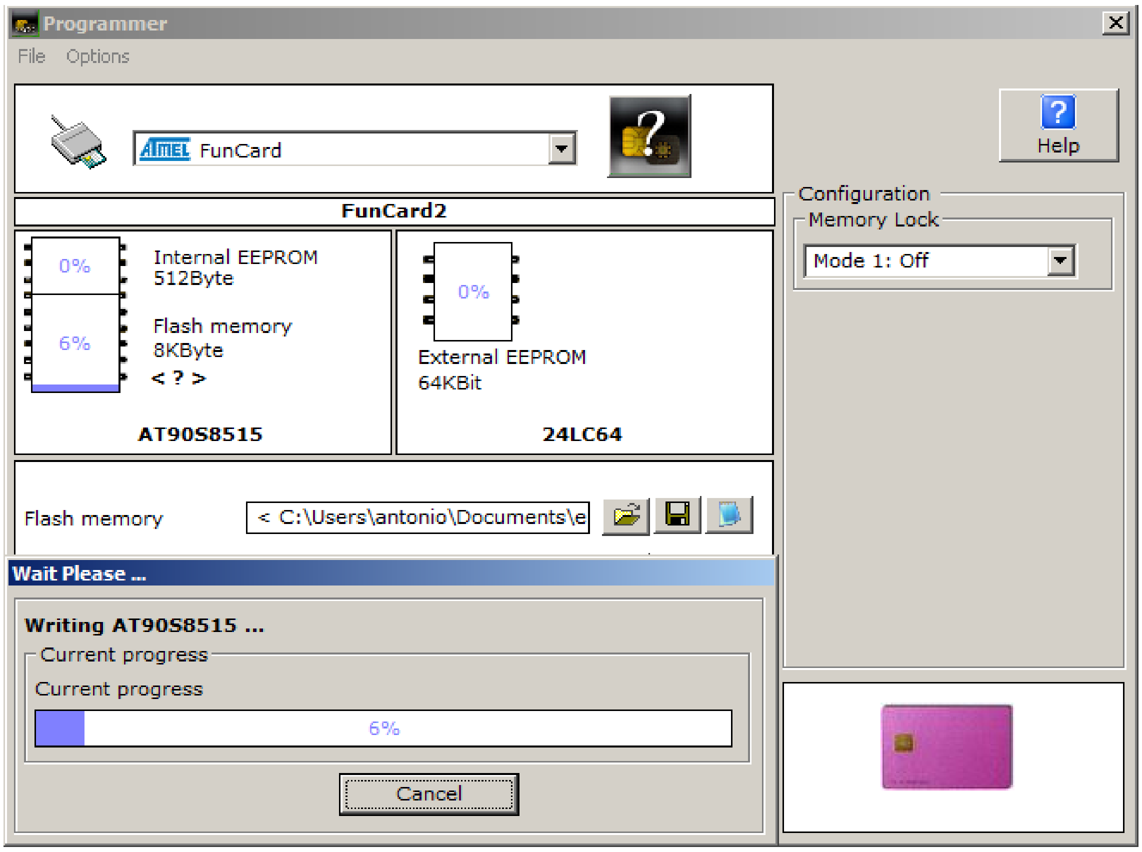
Flash memory
Internal EEPROM
External EEPROM

Dialog of the Card Programmer with the card recognized

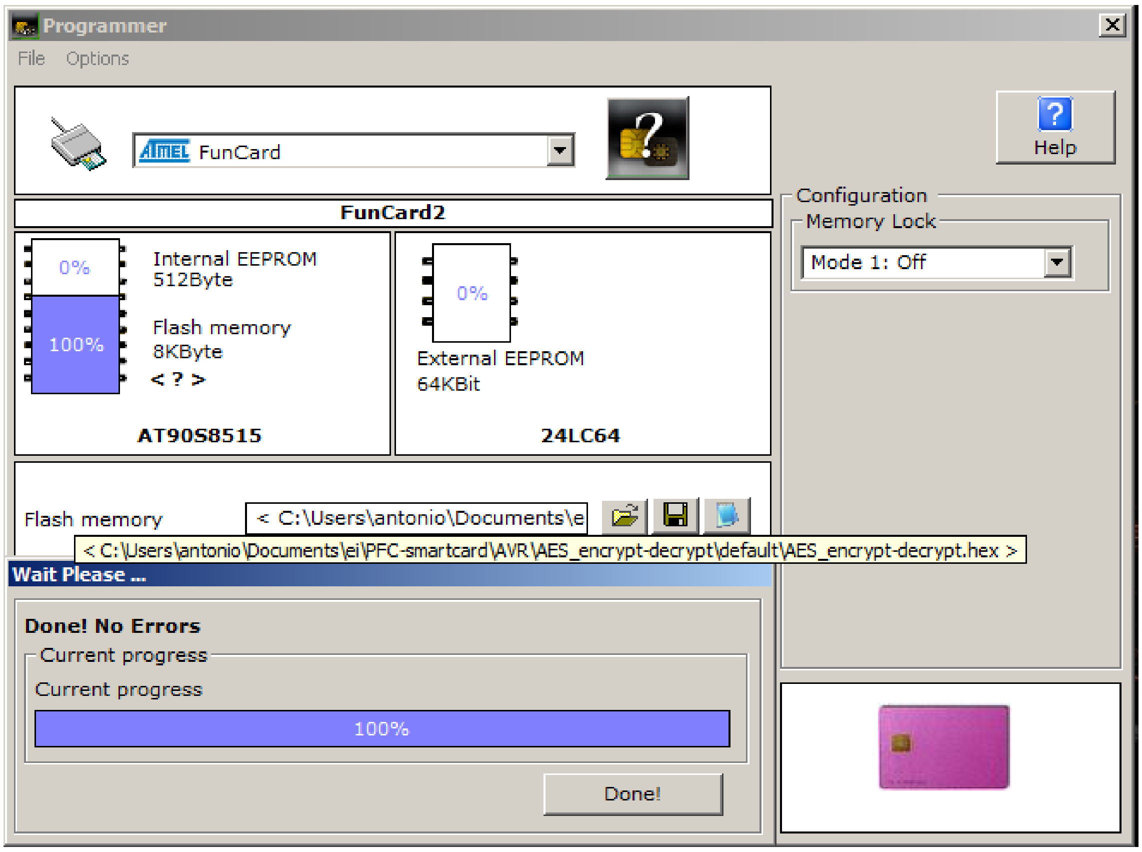
Select the files you wish to use for programming with the folder button, normally in the Flash memory. Click in the “Write” button. You can also specify in which part you wish to write, by clicking the side icon. In the next images appears the process of upload a hex file into the smart card.



Dialog of the Card Programmer with the file to write selected



Process of writing a file into the smart card



Process of upload a file into the smart card successfully

Installation guide

The installer file has to be downloaded from the website of Duolabs (http://www.duolabs.com) in the Download section.

Before starting the installation, the Dynamite +Plus programmer has not to be connected to the PC with a USB cable. The steps of the installation are the next.

- Follow the instructions of the setup program.
- Connect the Dynamite +Plus programmer with the USB cable to the PC.
- Execute Cas Studio and if it is necessary, change the interface language.

Depending of the operating system, you will need to some extra actions to complete the installation:

- Windows XP and older: Windows XP displays “Found New Hardware Wizard” window. Select “Install from a list or specific location”, click “Next”, select “Include this location in the search”, and then click “Browse” to open the “Browse for Folder” dialog. Search and select the c:\Programs\duolabs\Cas_Studioxxx\drivers folder you have created. Attention: this path may be different if you have specified a different folder during setup or if Windows is in other language. xxx stands for the release version of Cas Studio. Click OK, select “Next” and wait for the process to complete. Once the setup is completed, click “End”.
- Windows Vista: The drivers are automatically recognized and installed, any action is needed.

AVR Studio 4

AVR Studio 4 is the Integrated Development Environment (IDE) for developing 8-bit AVR applications in Windows NT/2000/XP/Vista/7 environments.

AVR Studio 4 provides a complete set of features including debugger supporting run control including source and instruction-level stepping and breakpoints; registers, memory and I/O views; and target configuration and management as well as full programming support for standalone programmers.

The features of AVR Studio 4 are described below:

- Integrated Assembler.
- Integrated Simulator.
- Integrates with GCC compiler plug-in.
- Support for all Atmel tools that support the 8-bit AVR architecture, including the AVR ONE!, JTAGICE mkI, JTAGICE mkII, AVR Dragon, AVRISP, AVR ISPmkII, AVR Butterfly, STK500 and STK600.
- AVR RTOS plug-in support.
- AT90PWM1 and ATtiny40 support.
- Command Line Interface tools updated with TPI support.
- Online help.

The AVR Studio offers a source code editor, project manager, assembler/compiler interface and a debugger to develop the applications.

AVR libc

The AVR libc package is a standard library for the C language which can be used in Atmel AVR 8-bit RISC microcontroller. This library provides the basic functions to use in the C language which are necessary in the most applications, for example, to work with strings (stdio.h header file).

AVR libc can be freely used and redistributed, provided the license conditions detailed in the project web.

The list of the modules supported by the library is described below:

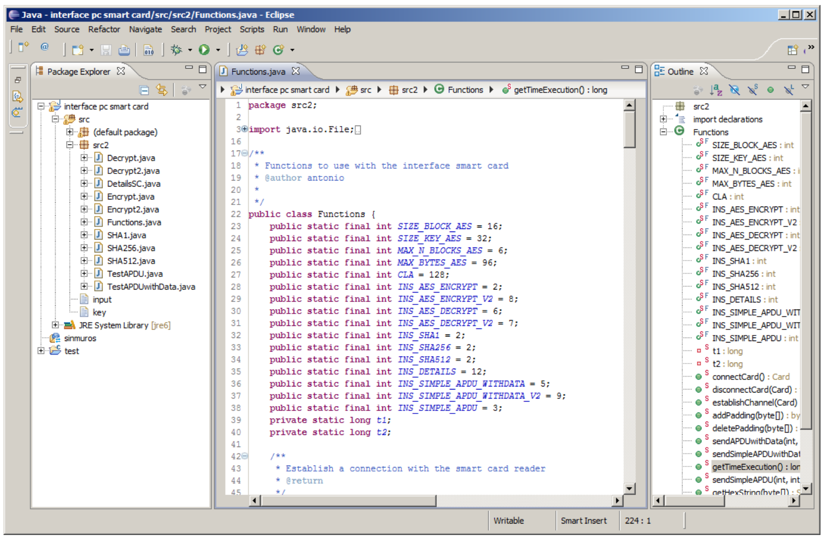
- Bootloader support utilities.
  - `#include <avr/io.h>`
  - `#include <avr/boot.h>`
- CRC computations.
  - `#include <avr/crc16.h>`
- EEPROM handling
  - `#include <avr/eeprom.h>`
- AVR device-specific IO definitions
  - `#include <avr/sfr.defs.h>`
- Program space string utilities
  - `#include <avr/io.h>`
  - `#include <avr/pgmspace.h>`

- Power management and sleep modes
    - `#include <avr/sleep.h>`
- Watchdog timer handling
    - `#include <avr/wdt.h>`
- Character operations
    - `#include <ctype.h>`
- System errors (errno)
    - `#include <errno.h>`
- Integer types
    - `#include <inttypes.h>`
- Mathematics
    - `#include <math.h>`
- Setjmp and Longjmp
- Standard IO facilities
    - `#include <stdio.h>`
- General utilities
    - `#include <stdlib.h>`
- Strings
    - `#include <string.h>`
- Interrupts and signals
    - `#include <signal.h>`
- Special function registers

Eclipse

Eclipse is a multi-language integrated development environment with a useful extensible plug-in system. It is oriented to write Java source code, but it can be used to develop applications in other languages as C, C++, COBOL, Python, Perl, PHP,… and it offers a multiple tools with the plug-in available to install in Ecliplse.

Released under the terms of the Eclipse Public License, Eclipse is free and open source software.


Java - interface pc smart card/src/src2/Functions.java - Eclipse
File Edit Source Refactor Navigate Search Project Scripts Run Window Help
Package Explorer
Functions.java
Outline
interface pc smart card ▸ src ▸ src2 ▸ Functions ▸ getTimeExecution() : long
Writable
Smart Insert
224 : 1

```java
package src2;

import java.io.File;

/**
 * Functions to use with the interface smart card
 * @author antonio
 *
 */
public class Functions {
    public static final int SIZE_BLOCK_AES = 16;
    public static final int SIZE_KEY_AES = 32;
    public static final int MAX_N_BLOCKS_AES = 6;
    public static final int MAX_BYTES_AES = 96;
    public static final int CLA = 128;
    public static final int INS_AES_ENCRYPT = 2;
    public static final int INS_AES_ENCRYPT_V2 = 8;
    public static final int INS_AES_DECRYPT = 6;
    public static final int INS_AES_DECRYPT_V2 = 7;
    public static final int INS_SHA1 = 2;
    public static final int INS_SHA256 = 2;
    public static final int INS_SHA512 = 2;
    public static final int INS_DETAILS = 12;
    public static final int INS_SIMPLE_APDU_WITHDATA = 5;
    public static final int INS_SIMPLE_APDU_WITHDATA_V2 = 9;
    public static final int INS_SIMPLE_APDU = 3;
    private static long t1;
    private static long t2;

    /**
     * Establish a connection with the smart card reader
     * @return
     */
```

Screen capture of Eclipse

Project analysis

Interface in Java with the smart card

An interface was developed in Java to communicate with the smart card with commands and responses APDU. Moreover, this interface includes extra methods to help with the common routine with the arrays of byte processing in Java.

The interface consists in series of classes to do a concrete cryptographic function. These classes are:

- Decrypt.java
- Decrypt2.java
- DetailsSC.java
- Encrypt.java
- Encrypt2.java
- Functions.java
- TestAPDU.java
- NotEncrypt2.java
- SHA1.java
- SHA256.java
- SHA512.java

In addition, there is a class called "Functions.java" where are implemented all the methods that the previous classes use.

The source code of the classes can be read in the appendix.

Encrypt class

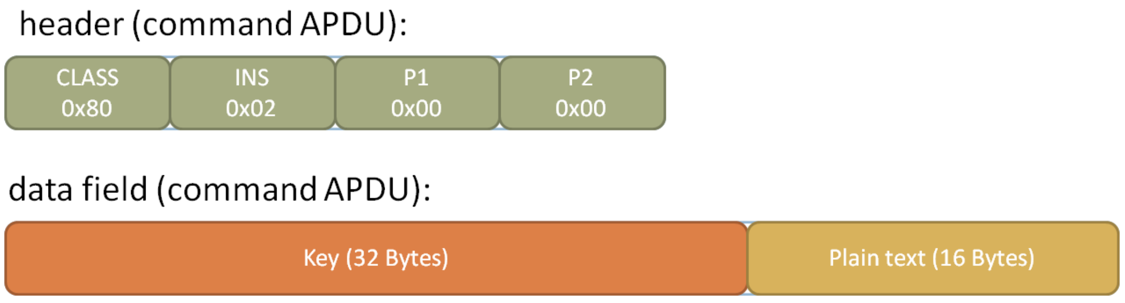
Briefly, this class calls the command "`do_AES_Encrypt`" implemented in the smart card. At the first, a communication channel is established with the smart card. For the AES encryption, it is necessary as parameter a key (32 Bytes) and the plain text (multiple of 16 Bytes).

The plain text is read from the file called "`C:/TMP/input`" and the key is read from the file called "`C:/TMP/key`". The method `readFile` (Functions.java) is used to read the data from a file and save it as array of bytes.

The plain text has to be multiple of 16 Bytes, so padding is calculated if it is necessary. The byte padding used is PKCS7 (this is described in RFC 3852). The padding is in whole bytes and the value of each added byte is the number of bytes is added. The number of bytes added will depends on the block boundary to which the message needs to be extended, in this case as maximum 16.

```
plainText = addPadding(plainText);
```

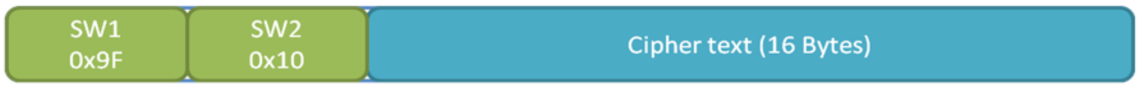
If the plain text is higher than 16 Bytes, each sequence of plain text of 16 Bytes is sent separately in the data field of its appropriate command APDU. In each command APDU, the key and the plain text are concatenated, because the key is not stored in the smart card and each time is needed, like is showed in the next image.



Structure of the command APDU sent to the smart card (Encrypt class)

If the plain text is not higher than 16 Bytes, the plain text and the key are concatenated in an array of bytes (`dataIn`) and a command APDU is sending to the smart card with this data.

The output encrypted is sent in a single response APDU and the data field of that response is stored in a buffer. When all the segments to encrypt have been received, the buffer contains the data encrypted. The structure of the response APDU is showed below.



Structure of the response APDU received from the smart card (Encrypt class)

To send the command APDU is necessary to use the method "`sendAPDUwithData`", in the manner shown below. To encrypt the data is necessary to select the command of the smart card "`do_AES_Encrypt`", with the Class 128 (`CLA`) and instruction 2 (`INS_AES_ENCRYPT`). The instruction parameter 1 and the instruction parameter 2 are not necessary and in the data field appears the array of bytes of data to send. Moreover, the channel of the communication with the smart card has to be sent on the method.

```
dataIn = concatenateArrayByte(key, plainTextBuf, plainTextBuf.length);
dataOutBuf = sendAPDUwithData(CLA, INS_AES_ENCRYPT, 0, 0, dataIn, channel);
```

Finally, the buffer that contains the data encrypted is written in a file ("`C:/TMP/output-encrypted`"). The method `writeFile` (Functions.java) is used to write the array of bytes in a file. After writing the file, the card is disconnected and the runtime of sent the command APDU is showed in the console.

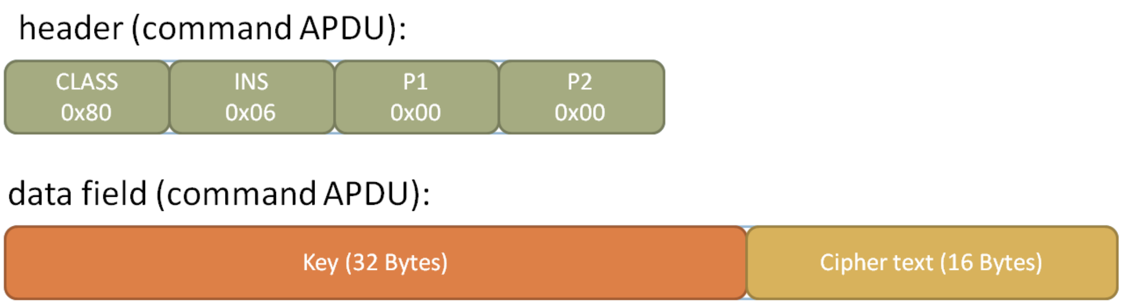
Decrypt class

This class is the responsible to decrypt a cipher text on the smart card. It works in a similar way than Encrypt class. At the first, a communication channel is established with the smart card. For the AES decryption, it is necessary as parameter the key (32 Bytes) and the cipher text.

The cipher text is read from the file "`C:/TMP/output-encrypted`" and the key is read from the file called "`C:/TMP/key`". The method readFile (Functions.java) is used to read the data from a file and to store it as array of Bytes.

The cipher text is multiple of 16 Bytes, so padding is not necessary to calculate. The problem of padding will appear after decryption process.

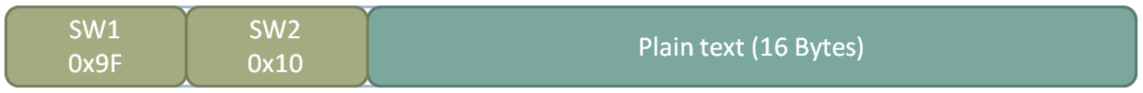
If the cipher text is higher than 16 bytes, each sequence of cipher text of 16 Bytes is sent separately in the data field of its appropriate command APDU. In each command, the key and the cipher text are concatenated, because the key is not stored in the smart card and each time is needed, like is showed in the next image.



Structure of the command APDU sent to the smart card (Decrypt class)

If the cipher text is not higher than 16 Bytes, the cipher text and the key are concatenated in an array of bytes (`dataIn`) and a command APDU is sending to the smart card with this data.

The output decrypted is sent in a single response APDU and the data field of that response is stored in a buffer. When all the segments to decrypt have been received, the buffer contains the clear data. The structure of the response APDU is showed below.



Structure of the response APDU received from the smart card (Decrypt class)

To send the command APDU is necessary to use the method "`sendAPDUwithData`". To decrypt the data is necessary to select the command of the smart card "`do_AES_Decrypt`", with the class 128 (`CLA`) and the instruction 6 (`INS_AES_DECRYPT`). The instruction parameter 1 and the instruction parameter 2 are not necessary and in the data field appears the array of bytes of data to decrypt. Moreover, the channel of the communication with the smart card has to be sent on the method.

```
dataOutBuf = sendAPDUwithData(CLA, INS_AES_DECRYPT, 0, 0, dataIn, channel);
```

The array of bytes "`dataOut`" contains the clear data, but it is compulsory to check if padding has been applied to the data encrypted and delete it if there is necessary.

```
dataOut = deletePadding(dataOut);
```

Finally, the buffer that contains the clear data is written in a file ("`C:/TMP/output-plainText`"). The method `writeFile` (Functions.java) is used to write the array of bytes in a file. After writing the file, the card is disconnected and the runtime of sent the command APDU is showed in the console.

Encrypt2 class

This class has the same functionality than Encrypt class. The difference is that in Encrypt class in one command APDU only is sent the key (32 Bytes) and the plain text (16 Bytes) concatenated, while in Encrypt2 class can send more than 16 Bytes of plain text in the data field of a command APDU. The maximum length of plain text to send is 224 Bytes because the key (32 Bytes) has to be sent too in the same command APDU and it is not possible to send more than 256 Bytes in the data field of a command APDU.

As a result the execution tests of Encrypt2, it is impossible to send more than 96 Bytes of plain text in one command APDU, because there is a memory overflow in the smart card (Unknown error 0x6f7).

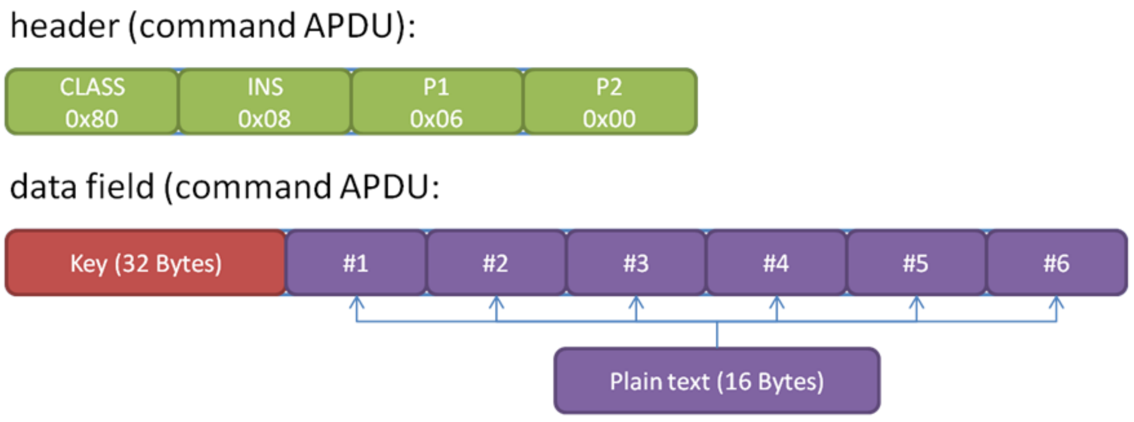
At the first, a communication channel is established with the smart card. For AES encryption, it is necessary as parameter a key (32 Bytes) and the plain text.

The plain text is read from the file called "`C:/TMP/input`" and the key is read from the file called "`C:/TMP/key`". The method `readFile` (Functions.java) is used to read the data from a file and save it as array of bytes.

The plain text has to be multiple of 16 Bytes, so padding is calculated if it is necessary. The byte padding used is PKCS7 (this is described in RFC 3852). The padding is in whole bytes and the value of each added byte is the number of bytes is added. The number of bytes added will depends on the block boundary to which the message needs to be extended, in this case as maximum 16.

```
plainText = addPadding(plainText);
```

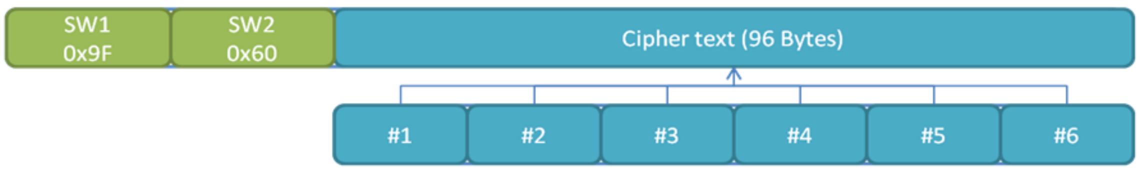
If the plain text is higher than 96 Bytes, each sequence of plain text of 96 Bytes is sent separately in the data field of its appropriate command APDU in blocks of 16 Bytes. In each command APDU, the key and the plain text are concatenated, because the key is not stored in the smart card and each time the key is requested. The structure of the command APDU is showed in the next image.



Structure of the command APDU sent to the smart card (Encrypt2 class)

If the plain text is not higher than 96 Bytes, the plain text and the key are concatenated in an array of bytes (`dataIn`) and a command APDU is sending to the smart card with this data.

The output encrypted is sent in a single response APDU and the data field of that response is stored in a buffer. When all the segments to encrypt have been received, the buffer contains the data encrypted. The structure of the response APDU is showed below.



Structure of the response APDU received from the smart card (Encrypt2 class)

To send the command APDU is necessary to use the method "`sendAPDUwithData`", in the manner shown below. To encrypt the data is necessary to select the command of the smart card "`do_AES_Encrypt2`", with the Class 128 (`CLA`) and instruction 8 (`INS_AES_ENCRYPT_V2`). The instruction parameter 1 contains the number of blocks of 16 bytes that are in the data field, the instruction parameter 2 is not necessary and in the data field appears the array of bytes of data to send. Moreover, the channel of the communication with the smart card has to be sent on the method.

```
dataIn = concatenateArrayByte(key, plainTextBuf, plainTextBuf.length);
int P1 = (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES;
try{
    dataOutBuf = sendAPDUwithData(CLA, INS_AES_ENCRYPT_V2, P1, 0, dataIn,
channel);
...
```

Finally, the array of bytes that contains the data encrypted is written in a file ("`C:/TMP/output-encrypted`"). The method `writeFile` (Functions.java) is used to write the array of bytes in a file. After writing the file, the card is disconnected and the runtime of sent the command APDU is showed in the console.

Decrypt2 class

This class has the same functionality than Decrypt class. The difference is that in Decrypt class in a command APDU only is sent the key (32 Bytes) and the 16 Bytes of cipher text concatenated in the data field, while in Decrypt2 class can be send more than 16 Bytes of cipher text in the data field of a command APDU. This is the same idea than in Encrypt2 class.

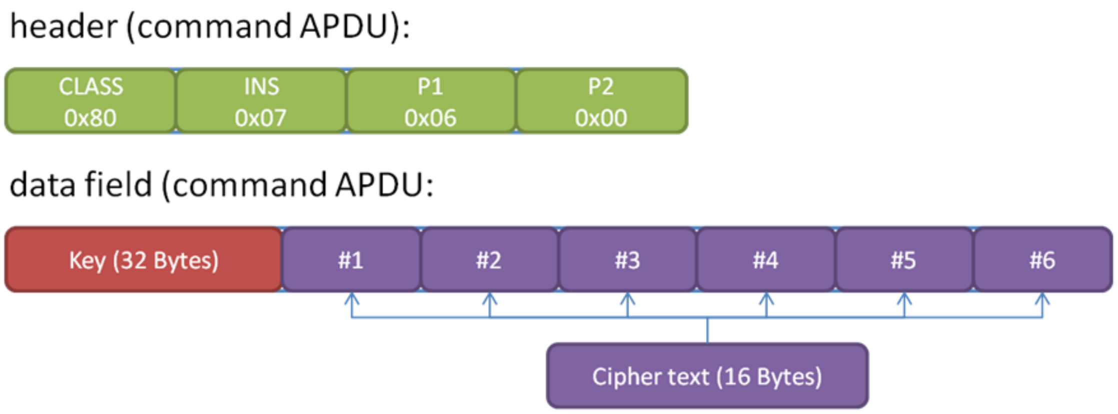
The maximum length of cipher text to send is 224 Bytes because the key (32 Bytes) has to be sent too in the same command APDU and it is not possible to send more than 256 Bytes in the data field of a command APDU.

As a result the execution tests of Decrypt2, it is impossible to send more than 96 Bytes of cipher text in one command APDU, because there is a memory overflow in the smart card (Unknown error 0x6f7).

At the first, a communication channel is established with the smart card. For AES decryption, it is necessary as parameter a key (32 Bytes) and the cipher text (at most 96 Bytes).

The cipher text is read from the file called "`C:/TMP/output-encrypted`" and the key is read from the file called "`C:/TMP/key`". The method `readFile` (Functions.java) is used to read the data from a file and save it as array of bytes.

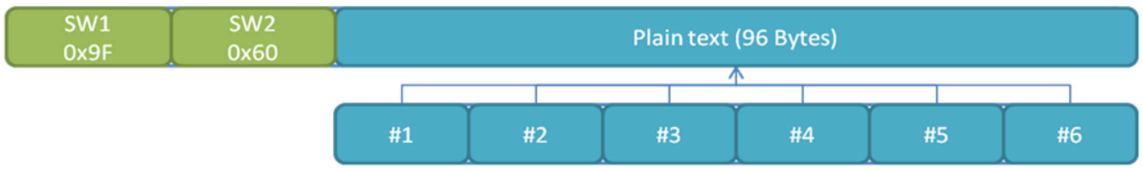
If the cipher text is higher than 96 Bytes, each sequence of cipher text of 96 Bytes is sent separately in the data field of its appropriate command APDU in blocks of 16 Bytes. In each command APDU, the key and the cipher text are concatenated, because the key is not stored in the smart card and each time it is requested. The structure of the command APDU is showed in the next image.



Structure of the command APDU sent to the smart card (Encrypt2 class)

If the cipher text is not higher than 96 Bytes, the cipher text and the key are concatenated in an array of bytes (`dataIn`) and a command APDU is sending to the smart card with this data.

The output decrypted is sent in a single response APDU and the data field of that response is stored in a buffer. When all the segments to decrypt have been received, the buffer contains the clear data. The structure of the response APDU is showed below.



Structure of the response APDU received from the smart card (Decrypt2 class)

To send the command APDU is necessary to use the method "`sendAPDUwithData`", in the manner shown below. To decrypt the data is necessary to select the comand of the smart card "`do_AES_Decrypt2`", with the Class 128 (`CLA`) and instruction 7 (`INS_AES_DECRYPT_V2`). The instruction parameter 1 contains the number of blocks of 16 bytes that are in the data field, the instruction parameter 2 is not necessary and in the data field appears the array of bytes of data to send. Moreover, the channel of the communication with the smart card has to be sent on the method.

```
dataIn = concatenateArrayByte(key, cipherTextBuf, cipherTextBuf.length);
int P1 = (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES;
try{
	dataOutBuf = sendAPDUwithData(CLA, INS_AES_DECRYPT_V2, P1, 0, dataIn,
channel);
```

The array of bytes "`dataOut`" contains the data decrypted, but it is compulsory to check if padding has been applied to the data encrypted and delete it if there is necessary.

```
dataOut = deletePadding(dataOut);
```

Finally, the buffer that contains the clear data is written in a file ("`C:/TMP/output-plainText`"). The method `writeFile` (Functions.java) is used to write the array of bytes in a file. After writing the file, the card is disconnected and the runtime of sent the command APDU is showed in the console.

SHA1 class

This class is responsible to do the digest of the input data with the algorithm SHA-1 implemented in the smart card. At the first, establish a channel communication with the smart card reader, and read the input data to digest from the file located in "`C:/TMP/input`".

To calculate the digest, an command APDU is sent to the smart card with the command "`sendAPDUwithData`" and selects the function of the smart card "`do_SHA1`", with the class 128 (`CLA`), the instruction 2 (`INS_SHA1`) and the length of the data to digest in the instruction parameter 1. The length of the data cannot be higher than 256 Bytes (maximum length of data field on command APDU). Moreover, the channel of the communication with smart card has to be sent on the method call. The commands to calculate digest are developed in other AVR project, so the instruction codes are not have to be different than the last.

Finally, the data returned (20 Bytes) on the previous method is the digest, which is showed on the terminal.

SHA256 class

This class calculates the digest of the input data with the algorithm SHA-256 implemented in the smart card. It works on the same way of SHA1 class, at the first, establish a channel communication with the smart card reader, and read the input data to digest from the file located in "`C:/TMP/input`".

To calculate the digest, an command APDU is sent to the smart card with the function "`sendAPDUwithData`" and selects the command "`do_SHA256`", with the class 128 (cla), the instruction 2 (`INS_SHA256`) and the length of the data to digest in the instruction parameter 1. The length of the data cannot be higher than 256 Bytes (maximum length of data field on command APDU). Moreover, the channel of the communication with smart card has to be sent on the method call.

Finally, the data returned (20 Bytes) on the previous method is the digest, which is showed on the terminal.

SHA512 class

This class calculates the digest of the input data with the algorithm SHA-512 implemented in the smart card. It works on the same way of SHA1 class and SHA256 class, except the changes on the number of instruction used 2 (`INS_SHA512`).

DetailsSC class

This class receives the details of the functions implemented in the smart card, using the command "`detailsApps`" of the smart card. At the first, a communication channel is established with the smart card with the functions "`connectCard`" and "`establishChannel`".

To select get the details of the smart card, a simple command APDU is sent. It is made up of a header with the class "`CLA`", the instruction "`INS_DETAILS`" and without any data in the data field. The method "`sendSimpleAPDUwithData`" is used to send the command APDU because an array of bytes is expected in the data field of the response APDU.

The data field of the response APDU contains the string coded in hexadecimal with UTF-8, so the data field has to be converted in a string to read it comfortably. At the end, the "`card`" card is released.

TestAPDU class

This class sends a simple command APDU using the method "`sendSimpleAPDU`" (Functions class). This class is used to verify that the smart card responses with a simple response APDU (only status word) and it is working.

The command APDU is made up of a header with the class "`CLA`" and the instruction "`INS_SIMPLE_APDU`". To send the command APDU to the smart card, the "`channel`" CardChannel is required.

Also, the runtime of the sending the command APDU is calculated and showed in the console. At the end, the "`card`" card is released.

NotEncrypt2 class

This class works on the same way than Encrypt2 class, but the difference is not using the same command in the smart card. The NotEncrypt2 class executes the command "`INS_AES_NOT_ENCRYPT_V2`" instead of "`INS_AES_ENCRYPT_V2`" as Encrypt2 class.

The main of this class is to calculate the runtime of a command APDU with the same structure than in Encrypt2 class and with the same response APDU expected, but the instruction "`INS_AES_NOT_ENCRYPT_V2`" do not execute the encryption functions. Without executing the encryption code, the runtime of the encryption code can be calculated by subtracting of the runtime of Encrypt2 class and NotEncrypt2 class.

At the first, a communication channel is established with the smart card. As in Encrypt2, the key (32 Bytes) and the plain text are read.

The plain text is read from the file called "`C:/TMP/input`" and the key is read from the file called "`C:/TMP/key`". The method `readFile` (Functions.java) is used to read the data from a file and save it as array of bytes.

The plain text has to be multiple of 16 Bytes, so padding is calculated if it is necessary. The byte padding used is PKCS7 (this is described in RFC 3852). The padding is in whole bytes and the value of each added byte is the number of bytes is added. The number of bytes added will depends on the block boundary to which the message needs to be extended, in this case as maximum 16.

```
plainText = addPadding(plainText);
```

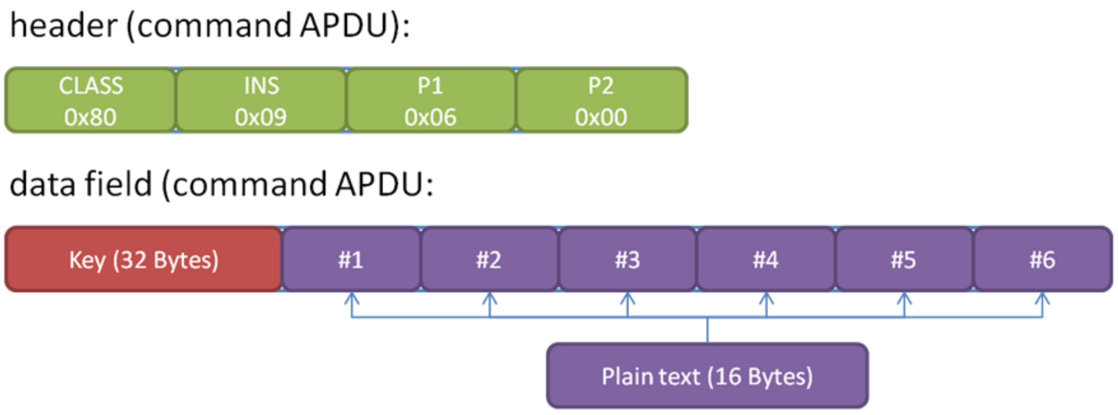
If the plain text is higher than 96 Bytes, each sequence of plain text of 96 Bytes is sent separately in the data field of its appropriate command APDU in blocks of 16 Bytes. In each command APDU, the key and the plain text are concatenated, because the key is not stored in the smart card and each time the key is requested.

If the plain text is not higher than 96 Bytes, the plain text and the key are concatenated in an array of bytes (`dataIn`) and a command APDU is sending to the smart card with this data.

The length of the data field of response APDU is compared with the data field of the command APDU to verify that has the same size and the output is correct.

To send the command APDU is necessary to use the method "`sendAPDUwithData`", in the manner shown below. The command APDU is made up of a header with the Class "`CLA`", the instruction "`INS_AES_NOT_ENCRYPT_V2`" (command "`do_AES_NotEncrypt2`" of the smart card), the instruction parameter 1 contains the number of blocks of 16 bytes that are in the data field, and the instruction parameter 2 is not necessary. The data field contains the array of bytes of data to send. Moreover, the channel of the communication with the smart card has to be sent on the method.

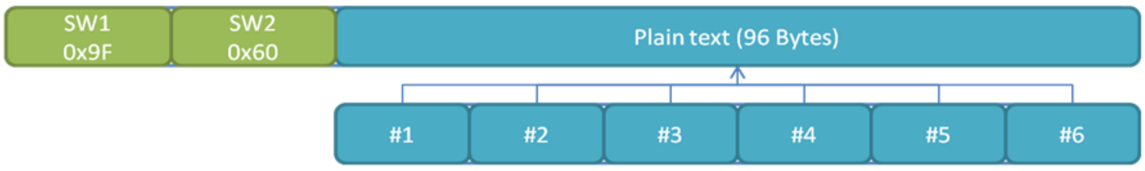
The structure of the command APDU sent is showed below.



Structure of the command APDU sent to the smart card (NotEncrypt2 class)

```
dataIn = concatenateArrayByte(key, plainTextBuf);
int P1 = (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES;
dataOut = sendAPDUwithData(CLA, INS_AES_NOT_ENCRYPT_V2, P1, 0, dataIn,
channel);
```

The array of bytes "`dataOut`" stores the plain text returned from the smart card and to check if the data received is correct, the length of the plain text sent and the plain text returned has to be the same.



Structure of the response APDU sent from the smart card (NotEncrypt2 class)

At the end, the card is released and the runtime of sent the command APDU without execute the encryption code is showed in the console.

NoEncrypt class

This class works on the same way than Encrypt class, but the difference is not using the same function in the smart card. The NoEncrypt class executes the command "`INS_AES_NOT_ENCRYPT`" instead of "`INS_AES_ENCRYPT`" as Encrypt class.

The main of this class is to calculate the runtime of a command APDU with the same structure than in Encrypt class and with the same response APDU expected, but the instruction "`INS_AES_NOT_ENCRYPT`" do not execute the encryption functions. Without executing the encryption code, the runtime of the encryption code can be calculated by subtracting of the runtime of Encrypt class and NoEncrypt class.

At the first, a communication channel is established with the smart card. As in Encrypt class, the key (32 Bytes) and the plain text are read.

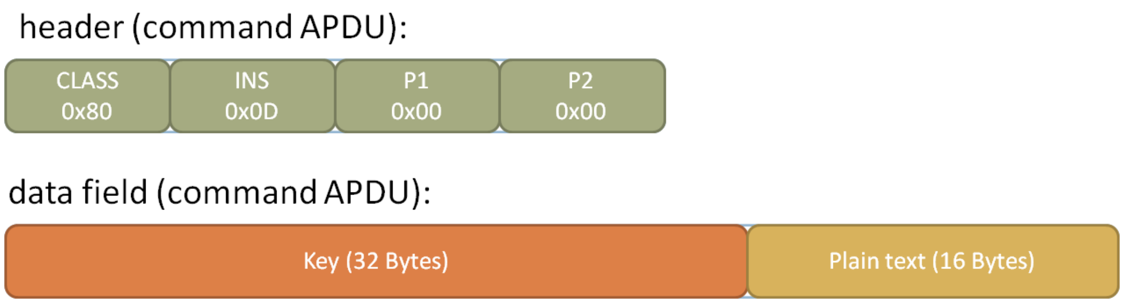
The plain text has to be multiple of 16 Bytes, so padding is calculated if it is necessary. The byte padding used is PKCS7 (this is described in RFC 3852). The padding is in whole bytes and the value of each added byte is the number of bytes is added. The number of bytes added will depends on the block boundary to which the message needs to be extended, in this case as maximum 16.

```
plainText = addPadding(plainText);
```

If the plain text is higher than 16 Bytes, each sequence of plain text of 16 Bytes is sent separately in the data field of its appropriate command APDU. In each command APDU, the key and the plain text are concatenated, because the key is not stored in the smart card and each time is needed.

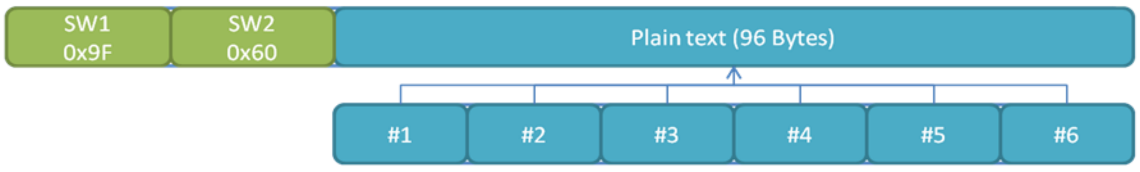
If the plain text is not higher than 16 Bytes, the plain text and the key are concatenated in an array of bytes (`dataIn`) and a command APDU is sending to the smart card with this data.

The structure of the command APDU sent is showed below.



Structure of the command APDU sent to the smart card (NoEncrypt class)

The length of the data field of response APDU is compared with the data field of the command APDU to verify that has the same size and the output is correct.



Structure of the response APDU sent from the smart card (NoEncrypt class)

To send the command APDU is necessary to use the method "`sendAPDUwithData`", in the manner shown below. The command APDU is made up of a header with the Class "`CLA`", the instruction "`INS_AES_NOT_ENCRYPT`" (command "`do_AES_NotEncrypt`" of the smart card), the instruction parameter 1 and the instruction parameter 2 are not necessary. The data field contains the array of bytes of data to send. Moreover, the channel of the communication with the smart card has to be sent on the method.

```
dataIn = concatenateArrayByte(key, plainTextBuf);
dataOutBuf = sendAPDUwithData(CLA, INS_AES_NOT_ENCRYPT, dataIn.length, 0,
dataIn, channel);
```

At the end, the card is released and the runtime of sent the command APDU without execute the encryption code is showed in the console.

Functions class

This class contains all the additional methods used by the previous classes. To keep the project clean, any class contains duplicated methods. Also, in Functions class are initialized the global variables used by the previous classes. The list of methods is as follows:

- `public static byte[] addPadding(byte[] plaintext)`
- `public static void byte[] concatenateArrayByte(byte[] key, byte[] plainText, int lengthPlainText)`
- `public static Card connectCard()`
- `public static byte[] deletePadding(byte[] dataOut)`
- `public static void disconnectCard(Card card)`
- `public static CardChannel establishChannel(Card card)`
- `public static long getTimeExecution()`
- `public static byte[] readFile(String fileName)`
- `public static sendAPDUwithData(int clas, int ins, int P1, int P2, byte[] dataIn, CardChannel channel) throws NullPointerException`
- `public static void sendSimpleAPDU(int cla, int ins, CardChannel channel)`
- `public static byte[] sendSimpleAPDUwithData(int clas, int ins, CardChannel channel) throws NullPointerException`
- `public static void writeFile(String fileName, byte[] data_write)`

The list of static final Integer variables is as follows:

- CLA
- INS_AES_DECRYPT
- INS_AES_DECRYPT_V2
- INS_AES_ENCRYPT
- INS_AES_ENCRYPT_V2
- INS_DETAILS
- INS_SHA1
- INS_SHA256
- INS_SHA512
- INS_SIMPLE_APDU
- INS_SIMPLE_APDU_WITHDATA
- INS_AES_NOT_ENCRYPT_V2
- MAX_BYTES_AES
- MAX_N_BLOCKS_AES
- SIZE_BLOCK_AES
- SIZE_KEY_AES

The list of private static long variables:

- t1
- t2

The variable `CLA` defines the number of class of the instructions on the smart card, which always have the number 128. The variables `INS_AES_DECRYPT`, `INS_AES_DECRYPT_V2`, `INS_AES_ENCRYPT`, `INS_AES_ENCRYPT_V2` belong to the numbers of instruction of each encryption and decryption functions implemented in the smart card. The variable `INS_DETAILS` belong to the instruction to get details about the functions implemented in the smart card. The variables `INS_SHA1`, `INS_SHA256` and `INS_SHA` belong to the functions to digest data in the smart card, but they have never implemented. The variables `INS_SIMPLE_APDU`, `INS_SIMPLE_APDU_WITHDATA`, `INS_AES_NOT_ENCRYPT_V2`, `INS_AES_NOT_ENCRYPT` belong to testing functions in the smart card used in the development of the previous cryptographic functions. Ultimately, the variables `MAX_BYTES_AES`, `MAX_N_BLOCKS_AES`, `SIZE_BLOCK_AES` and `SIZE_KEY_AES` are used to simplify the constant values used in the encryption and decryption process.

The description of the methods of the Functions class appears below:

addPadding method

This method adds padding to a array of bytes until the array would be multiple of `SIZE_BLOCK_AES`. If "`plainText`" is not multiple of `SIZE_BLOCK_AES`, the method returns a new array with a length multiple of `SIZE_BLOCK_AES` and with the padding added. If the padding is not necessary to add, the method returns the variable "`plainText`".

```
public static byte[] addPadding(byte[] plainText){
        int N = plainText.length / SIZE_BLOCK_AES;
        int resto = plainText.length % SIZE_BLOCK_AES;
        byte[] plainText2 = null;
        if(resto != 0){ // add padding
                int n2 = (N+1)*SIZE_BLOCK_AES;
                resto = n2 % plainText.length;
                plainText2 = new byte[n2];
                for(int i=0; i<plainText2.length; i++){
                        if(i<plainText.length){
                                plainText2[i] = plainText[i];
                        }else{
                                plainText2[i] = (byte) resto;
                        }
                }
        }else{
                plainText2 = plainText;
        }
        return plainText2;
}
```

concatenateArrayByte method

This method returns a concatenated array of bytes from two arrays of bytes called "`key`" and "`plainText`". It is a simple routine to do not dirty the other classes if an array of bytes concatenation is necessary.

```
public static byte[] concatenateArrayByte(byte[] key, byte[] plainText){
        byte[] ret = new byte[SIZE_KEY_AES+plainText.length];
        int i;
        for(i=0; i<SIZE_KEY_AES; i++) ret[i] = key[i];
        for(i=SIZE_KEY_AES; i<(SIZE_KEY_AES+plainText.length); i++) ret[i] =
plainText[i-SIZE_KEY_AES];
        return ret;
}
```

connectCard method

The method "`connectCard`" establish a connection with the smart card reader, the initialization of the variables necessary (Card class) to start to work with the smart card. At first, the terminals supported by the default TerminalFactory are collected in a list. The connection is established with the first terminal of the list and with the protocol used is the block-oriented T=1 protocol.

The details of the card connected are showed in the console and the list of the terminals supported by the default TerminalFactory are also showed by the console, that are the terminals connected in the computer.

The method returns a Card object with the connection necessary to transmit command APDU and receive response APDU.

```
public static Card connectCard(){
        TerminalFactory factory;
        List<CardTerminal> terminals;
        CardTerminal terminal;
        Card card = null;
        try{
                factory = TerminalFactory.getDefault();
                terminals = factory.terminals().list();
                System.out.println("Terminals: "+terminals);
                terminal = terminals.get(0);
                card = terminal.connect("T=1");
                System.out.println("card :"+card);
        }catch(CardException ex1){
                System.out.println(ex1.getMessage());
                System.exit(-1);
        }
        return card;
}
```

deletePadding method

The "`deletePadding`" method deletes the padding in the array of bytes called "`dataOut`" if there is padding in the array. If there is not padding in the array, the method returns the same "`dataOut`" array. At first, the method checks if there are padding in "`dataOut`" and if there is, below it is deleted.

```
public static byte[] deletePadding(byte[] dataOut){
        boolean pad = false;
        int last = (int) dataOut[dataOut.length-1];
        if(last>0 && last<SIZE_BLOCK_AES){
                pad = true;
                for(int i=(dataOut.length-last); i<dataOut.length && pad; i++){
                        if(dataOut[i] != last) pad = false;
                }
        }
        if(!pad){
                return dataOut;
        }else{
                int N = dataOut.length - last;
                byte[] dataOut2 = new byte[N];
                for(int i=0; i<N; i++){
                        dataOut2[i] = dataOut[i];
                }
                return dataOut2;
        }
  }
```

disconnectCard method

The "`disconnectCard`" method disconnects the "`card`" to release the connection established before). If the disconnect operation fails, a CardException is caught and the message is showed in the console.

```
public static void disconnectCard(Card card){
      try{
            card.disconnect(false);
      }catch(CardException ex1){
            System.out.println(ex1.getMessage());
            System.exit(-1);
      }
}
```

establishChannel method

The "`establishChannel`" returns the "`channel`" (CardChannel object) for the basic logic channel of the "`card`". The basic logical channel has a channel number of 0.

```
public static CardChannel establishChannel(Card card){
      CardChannel channel = null;
      channel = card.getBasicChannel();
      return channel;
}
```

getTimeExecution method

The "`getTimeExecution`" method returns the runtime of the command APDU sent to the smart card. The value is calculated from the subtraction global variables "`t2`" with "`t1`". The value of the previous variables is set up in the methods "`sendAPDUwithData`", "`sendSimpleAPDU`" and "`sendSimpleAPDUwithData`".

```
public static long getTimeExecution()
      return t2-t1;
}
```

readFile method

The "`readFile`" method returns an array of bytes resulting of the read of a file located in the "`fileName`" path. This method creates the objects "`File`" and "`FileInputStream`" to read the content of the file. Previously of read, the array of bytes is initialized with the length of the content of the file. The exceptions "`FileNotFoundException`", "`IOException`" and "`Exception`" are caught and them respective error messages are showed in the console. If finally there is not any data read, the method returns null.

```
public static byte[] readFile(String fileName){
      File file;
      FileInputStream fis;
      byte[] data_read = null;
```

```
        int read = -1;

        try{
                file = new File(fileName);
                fis = new FileInputStream(file);
                int N = fis.available();
                data_read = new byte[N];
                read = fis.read(data_read);
        }catch(FileNotFoundException ex1){
                System.out.println(ex1.getMessage());
        }catch(IOException ex2){
                System.out.println(ex2.getMessage());
        }catch(Exception ex3){
                System.out.println(ex3.getMessage());
        }
        if(read != -1){
                return data_read;
        }else{
                return null;
        }
}
```

sendAPDUwithData method

The "`sendAPDUwithData`" method is the most important method of the Functions class. This method is the responsible to send a command APDU and return an array of bytes from the data field of the response APDU obtained. The global variables "`t1`" and "`t2`" are initialized to calculate the runtime of the command APDU.

The command APDU is built with the next values in the header: the variable "`clas`" defines the class, the variable "`ins`" defines the instruction, the variables "`P1`" and "`P2`" defines the parameters. The array of bytes "`dataIn`" constitutes the data field. The variable "`channel`" is necessary to send the command APDU to the smart card.

If the status word 1 of the response APDU is not 159 (0x9F), the "`NullPointerException`" exception is thrown. The "CardException" exception is caught if the card operation failed, and its message is showed in the console.

```
public static byte[] sendAPDUwithData(int clas, int ins, int P1, int P2,
byte[] dataIn, CardChannel channel) throws NullPointerException{
        byte[] dataOut = null;
        ResponseAPDU r = null;
        CommandAPDU c = null;
        try{
                c = new CommandAPDU(clas, ins, P1, P2, dataIn);
                t1 = System.currentTimeMillis();
                r = channel.transmit(c);
                t2 = System.currentTimeMillis();
                dataOut = r.getData();
        }catch(CardException ex1){
                System.out.println("Error sendAPDUwithData "+ex1.getMessage());
                System.exit(-1);
        }
        if(r.getSW1() != 159){
                NullPointerException ex2 = new NullPointerException("Error with
the APDU response, data returned invalid");
                throw ex2;
        }else{
                return dataOut;
        }
}
```

sendSimpleAPDU method

The “sendSimpleAPDU” method sends a simple command APDU to the smart card. The command APDU has not any data on its data field, so it is built with the variables “cla” that defines the class and “ins” that defines the instruction in the header. The variable “channel” is necessary to send the command APDU to the smart card.

The global variables “t1” and “t2” are initialized to calculate the runtime of the command APDU.

In the console is showed a string representation of the response APDU obtained. The “CardException” exception is caught if the card operation failed, and its message is showed in the console.

```
public static void sendSimpleAPDU(int cla, int ins, CardChannel channel){
	try{
		CommandAPDU c = new CommandAPDU(cla, ins, 0, 0);
		t1 = System.currentTimeMillis();
		ResponseAPDU r = channel.transmit(c);
		t2 = System.currentTimeMillis();
		System.out.println(r);
	}catch(CardException ex1){
		System.out.println(ex1.getMessage());
		System.exit(-1);
	}
}
```

sendSimpleAPDUwithData method

The “sendSimpleAPDUwithData” method works on the same way than “sendAPDUwithData”. The method is the responsible to send a command APDU and return an array of bytes from the data field of the response APDU obtained. The global variables “t1” and “t2” are initialized to calculate the runtime of the command APDU.

The difference between this method and “sendAPDUwithData” is that the command APDU is built only with the variable “clas” (defines the class in the header) and with the variable “ins” (defines the instruction in the header).

If the status word 1 of the response APDU is not 159 (0x9F), the “NullPointerException” exception is thrown. The “CardException” exception is caught if the card operation failed, and its message is showed in the console.

```
public static byte[] sendSimpleAPDUwithData(int clas, int ins, CardChannel
channel) throws NullPointerException{
	byte[] dataOut = null;
	ResponseAPDU r = null;
	CommandAPDU c = null;
	try{
		c = new CommandAPDU(clas, ins, 0, 0);
		t1 = System.currentTimeMillis();
		r = channel.transmit(c);
		t2 = System.currentTimeMillis();
		System.out.println(r);
		dataOut = r.getData();
	}catch (CardException  ex1) {
		System.out.println("Error sendAPDUwithData "+ex1.getMessage());
		System.exit(-1);
	}
	if(r.getSW1() != 159){
		NullPointerException ex2 = new NullPointerException("Error with
the APDU response, data returned invalid");
		throw ex2;
	}else{
		return dataOut;
	}
}
```

writeFile method

The "writeFile" method writes the "data_write" array of bytes in the file located in the "fileName" path. This method creates the objects "File" and "FileOutputStream" to write the content of the array and in the variable "fis" is written the entire array. The exceptions "FileNotFoundException" (the file to open denoted by a specified pathname has failed), "IOException" (if an I/O error occurs) and "Exception" are caught and them respective error messages are showed in the console.

```
public static void writeFile(String fileName, byte[] data_write){
	File file;
	FileOutputStream fis;

	try{
		file = new File(fileName);
		fis = new FileOutputStream(file);
		fis.write(data_write);
	}catch(FileNotFoundException ex1){
		System.out.println(ex1.getMessage());
	}catch(IOException ex2){
		System.out.println(ex2.getMessage());
	}catch(Exception ex3){
		System.out.println(ex3.getMessage());
	}
}
```

Cryptographic applications on the smart card (AES encryption and decryption)

The code of "AES encryption and decryption" was developed with the integrated development environment "AVR Studio" (version 4.17). For the development, a project was created with the name "AES encryption and decryption".

The organization of the project appears below. The description of each section and the subsections appears later the organization.

- Source files
  - aes32.c
  - functions_smartcard.c
  - main.c
- Header files
  - aes32.h
  - functions_smartcard.h
  - globals.h

  - T1_Comm_Lib.h
- External dependencies
  - common.h
  - deprecated.h
  - fuse.h
  - inttypes.h
  - io.h
  - iomega163.h
  - libT1_Comm_Lib.a
  - lock.h
  - portpins.h
  - sfr_defs.h
  - stdint.h
  - string.h
  - version.h
- Other files
  - AES_encrypt-decrypt.lss
  - AES_encrypt-decrypt.map

The external library used is "libT1_Comm_Lib.a" to communicate with the smart card reader using the commands APDU and responses APDU. The configuration of the compilers is for the Device atmega163 (smart card used in the project). The configuration file of the project is attached in the appendix.

The images of the command APDU and response APDU used in each function appear previously, in the section "Interface in Java with the smart card".

In the structure "`response_APDU`" appears the variables "`LEN`" and "`LE`". The variable "`LEN`" refers to the length of the prologue field of the command APDU and its value is always the length of the data field plus the 2 Bytes of the status word. The variable "`LE`" refers to the length of the data field.

The source code of the AVR project appears in the appendix.

Header files

aes32.h

The header file "aes32.h" contains the declaration of the functions available to encrypt and decryption data in the smart card. Also, it contains the structure ("*struct*") "`aes256_context`" used to store the key in memory for the encryption or decryption process.

functions_smartcard.h

This header file contains the declaration of the functions written in "`functions_smartcard.c`".

globals.h

This header file contains some global constants like true, false, null, return codes. The global type definitions are also included, e.g. the constants used in functions of "`functions_smartcard.c`" the structure "`command_APDU`" and the structure "`response_APDU`". The latter structures are very important because there are fundamental in the project and they are widely used.

T1_Comm_Lib.h

This header file contains the declaration of the functions implemented in the external library "`libT1_Comm_Lib.a`". The functions declared are fundamental because they are responsible of the management of the commands APDU and responses APDU.

```
void send_ATR(void);
int request_extended_BWT(response_APDU *send_APDU , char extension_factor);
unsigned char receive_APDU(command_APDU *received_APDU);
void send_APDU(response_APDU *send_APDU);
```

Source files

aes32.c

The file "aes32.c" contains the source code of the byte-oriented implementation of the algorithm Advanced Encryption Standard (AES-32) for encrypt and decrypt blocks of 16 Bytes with a key of 32 Bytes. The mode of encryption used is the electronic codebook (ECB), where the message is divided into blocks and each block is encrypted separately.

The functions available to use are the next.

```
void aes256_init(aes256_context *, uint8_t * /* key */);
void aes256_done(aes256_context *);
void aes256_encrypt_ecb(aes256_context *, uint8_t * /* plaintext */);
void aes256_decrypt_ecb(aes256_context *, uint8_t * /* cipertext */);
```

functions_smartcard.c

In this file appears the source code of the applications implemented to encrypt and decrypt data with the AES-32 algorithm in the smart card. In this source file, the arguments of the functions are a pointer of a command APDU and a pointer of a response APDU, because the management of sending and receiving of the command APDU is in the file "`main.c`".

The functions developed are:

do_AES_Encrypt function

This function is the responsible of the simple encryption and it can encrypt 16 Bytes of input plain text with 32 Bytes of a key returning 16 Bytes of cipher text as output.

The class of the command APDU expected is" 0x80" and the instruction is "0x02". The parameter 1 and the parameter 2 are not required to have any value. The data field of the command APDU has to contain 16 Bytes of plain text to encrypt and the 32 Bytes of the key. The structure of the command APDU appears in the subsection "Encrypt class".

At the first, the array of unsigned integer "key" and "buf" are created with a length of 32 Bytes and 16 Bytes respectively. The key used in the encryption is stored in the array "key" and it is composed of the first 32 Bytes of the data field of the command APDU. The plain text to encrypt is stored in the array "buf" and it is composed of the next 16 Bytes (after the first 32 Bytes).

The structure "aes256_content" has to be initialized with the key and after that, it is possible to encrypt the plain text with the function "aes256_encrypt_ecb". The function "aes256_done" release the key from the structure "aes256_content".

```
aes256_init(&ctx, key);
aes256_encrypt_ecb(&ctx, buf);
aes256_done(&ctx);
```

Finally, in the data field of the response APDU is copied the array "buf", which contains the cipher text obtained. The variable "LE" contains the value of the size of a block ("SIZE_BLOCK_AES") and the variable "LEN" contains the sum of the value "LE" and the size of a status word (2). The header is composed with the value 0x9F (Success) in the status word 1 and the value 0x10 in the status word 2 (number of bytes of data available to read in the data field). The structure of the response APDU appears in the subsection "Encrypt class".

test_command function

This function sends a simple response APDU when it is selected.

The class of the command APDU expected is "0x80" and the instruction is "0x03". The parameter 1 and the parameter 2 are not required to have any value.

The response APDU "resp_APDU" is made up of a header with the next values: "0x90" in the status word 1, "0x00" in the status word 2, "SIZE_SW_RESPONSE" in the variable "LEN" and "0" in the variable "LE" (there is not any bytes of data available to read in the data field). The status word of the response means that the command has been executed without error.

The use of this function is to verify that the smart card is working and response with a simple command APDU.

test_timer0 function

This function measures the number of clock cycles necessary to copy the data available in the command APDU in a local array of unsigned integer.

The class of the command APDU expected is" 0x80" and the instruction is "0x04". The parameter 1 contains the length of the data field. The parameter 2 is not required to have any value. The data field of the command APDU contains the data to copy in the local array.

If the parameter 1 of the header does not contains a number higher than 0 and smaller than 15, the response APDU "resp" with the status word "6A 68" (referenced data not found) is returned and the function ends.

At the first, the timer 0 is initialized with the next values.

```
overflow = 0;
DDRB = 0xFF;
TCNT0 = 0;
```

```
TCCR0 = 1;
```

The variable "overflow" checks if a overflow happens between in sampling clock cycles. The register "DDRB" is set up to use all the pins on "PORTB" (8-bit bi-directional I/O port) for output. The register "TCNT0" (timer/counter 0 value) is set up to 0 to start value of timer/counter0. The register "TCCR0" (timer/counter 0 control register) is set up to 1 to set the oscillator frequency.

The variables "cal_ref_timer" and "cal_ref_timer2" contains the clock cycles between the main loop.

```
cal_ref_timer = TCNT0;
for(i=0; i<N; i++){
        buffer[i] = (*com_APDU).data_field[i];
}
cal_ref_timer2 = TCNT0;
```

It is necessary to check if the register "TIFR" (timer interrupt flag register) is set to avoid an overflow in the measure of the values of the timer/counter 0.

```
if((TIFR&0x01) == 0x01){
        overflow = 1;
        cal_ref_timer2 = 0x00;
}
```

The status word 1 of the response APDU "resp_APDU" is "SW1_SUCCESS" and the status word 2 is the amount of bytes in the data field, in this case 3. The data field consists of:

- First byte: value of "cal_ref_timer".
- Second byte: value of "cal_ref_timer2".
- Third byte: value of "overflow". If this byte is set, an overflow happened in the measure of the previous values and they are incorrect.

test_command_withData function

This function copies the data field of the command APDU "com_APDU" in the data field of the response APDU "resp_APDU".

The class of the command APDU expected is "0x80" and the instruction is "0x05". The parameter 1 contains the length of the data field. The parameter 2 is not required to have any value. The data field of the command APDU contains the data to copy in "resp_APDU".

The loop copies the data between the command APDU to response APDU.

```
for(int i=0; i<N; i++){
        (*resp_APDU).data_field[i] = (*com_APDU).data_field[i];
}
```

The status word 1 of the response APDU "resp_APDU" is "SW1_SUCCESS" and the status word 2 is the amount of bytes in the data field, in this case the parameter 1 of the command APDU. The data field contains the data copied. The values "LE" and "LEN" are set up with the length of the data field and the length of the data field with the header, respectively.

do_AES_Decrypt function

This function is the responsible of the simple decryption and it can decrypt 16 Bytes of input cipher text with 32 Bytes of a key returning 16 Bytes of clear text as output.

The class of the command APDU expected is" 0x80" and the instruction is "0x06". The parameter 1 and the parameter 2 are not required to have any value. The data field of the command APDU has to contain 16 Bytes of cipher text to decrypt and the 32 Bytes of the key.

At the first, the array of unsigned integer "key" and "buf" are created with a length of 32 Bytes and 16 Bytes respectively. The key used in the decryption is stored in the array "key" and it is composed of the first 32 Bytes of the data field of the command APDU. The cipher text to decrypt is stored in the array "buf" and it is composed of the next 16 Bytes (after the first 32 Bytes). In the subsection "Decrypt class" appears the structure of the command APDU and response APDU.

The structure "aes256_content" has to be initialized with the key and after that, it is possible to decrypt the cipher text with the function "aes256_decrypt_ecb". The function "aes256_done" release the key from the structure "aes256_content".

```
aes256_init(&ctx, key);
aes256_decrypt_ecb(&ctx, buf);
aes256_done(&ctx);
```

Finally, in the data field of the response APDU is copied the array "buf", which contains the clear text obtained. The variable "LE" contains the value of the size of a block ("SIZE_BLOCK_AES") and the variable "LEN" contains the sum of the value "LE" and the size of a status word (2). The header is composed with the value 0x9F (Success) in the status word 1 and the value 0x10 in the status word 2 (number of bytes of data available to read in the data field).

do_AES_Decrypt2 function

This function can decrypt until 96 Bytes (in blocks of 16 Bytes) of input cipher text with 32 Bytes of a key returning the clear text as output.

The class of the command APDU expected is" 0x80" and the instruction is "0x07". The parameter 1 contains the number of blocks of 16 Bytes with cipher text (as maximum 6).The parameter 2 is not required to have any value. The data field of the command APDU contains the cipher text to decrypt in blocks of 16 Bytes (as maximum 96 Bytes) and 32 Bytes of the key. In the subsection "Decrypt2 class" appears the structure of the command APDU and response APDU.

The variable "N" contains the number of blocks to decrypt indicated in the parameter 1.

At the first, the array of unsigned integer "key" and "buf" are created with a length of 32 Bytes and the value of "N", respectively. The key used in the decryption is stored in the array "key" and it is composed of the first 32 Bytes of the data field of the command APDU.

If the number of blocks to decrypt ("N") is higher than the maximum, the response APDU "resp_APDU" with the status word "6A 68" (referenced data not found) is returned and the command ends.

The structure "aes256_content" has to be initialized with the key. After that, appears the main loop that iterates depending the number of blocks to decrypt. In each iteration, the block is copied in "buf", it is decrypted with the function "aes256_decrypt_ecb" and the clear output "buf" is copied in the data field of the response APDU "resp_APDU". After all the blocks are decrypted, the function "aes256_done" release the key from the structure "aes256_content".

```
aes256_init(&ctx, key);
for(i=0; i<N; i++){
	for(j=0; j<SIZE_BLOCK_AES; j++){
		buf[j] = 
(*com_APDU).data_field[SIZE_KEY_AES+(i*SIZE_BLOCK_AES)+j];
	}
	aes256_decrypt_ecb(&ctx, buf);
	for(j=0; j<SIZE_BLOCK_AES; j++){
		(*resp_APDU).data_field[(i*SIZE_BLOCK_AES)+j] = buf[j];
	}
}
aes256_done(&ctx);
```

Finally, the header of the response APDU "resp_APDU" is composed with the value 0x9F (Success) in the status word 1 and the value 0x10 in the status word 2 (number of bytes of data available to read in the data field). The variable "LE" contains the value of the size of the data field ("SIZE_BLOCK_AES*N") and the variable "LEN" contains the sum of the value "LE" and the size of a status word (2). The data field was filled in the last loop.

do_AES_Encrypt2 function

This function can encrypt until 96 Bytes (in blocks of 16 Bytes) of input plain text with 32 Bytes of a key returning the clear text as output.

The class of the command APDU expected is" 0x80" and the instruction is "0x08". The parameter 1 contains the number of blocks of 16 Bytes with plain text (as maximum 6).The parameter 2 is not required to have any value. The data field of the command APDU contains the plain text to encrypt in blocks of 16 Bytes (as maximum 96 Bytes) and 32 Bytes of the key. In the subsection "Encrypt2 class" appears the structure of the command APDU and response APDU.

The variable "N" contains the number of blocks to encrypt indicated in the parameter 1.

At the first, the array of unsigned integer "key" and "buf" are created with a length of 32 Bytes and the value of "N", respectively. The key used in the encryption is stored in the array "key" and it is composed of the first 32 Bytes of the data field of the command APDU.

If the number of blocks to encrypt ("N") is higher than the maximum, the response APDU "resp_APDU" with the status word "6A 68" (referenced data not found) is returned and the command ends.

The structure "aes256_content" has to be initialized with the key. After that, appears the main loop that iterates depending the number of blocks to encrypt. In each iteration, the block is copied in "buf", it is encrypted with the function "aes256_encrypt_ecb" and the cipher output "buf" is copied in the data field of the response APDU "resp_APDU". After all the blocks are encrypted, the function "aes256_done" release the key from the structure "aes256_content".

```
aes256_init(&ctx, key);
for(i=0; i<N; i++){
	for(j=0; j<SIZE_BLOCK_AES; j++){
```

```
                buf[j] =
(*com_APDU).data_field[SIZE_KEY_AES+(i*SIZE_BLOCK_AES)+j];
        }
        aes256_encrypt_ecb(&ctx, buf);
        for(j=0; j<SIZE_BLOCK_AES; j++){
                (*resp_APDU).data_field[(i*SIZE_BLOCK_AES)+j] = buf[j];
        }
}
aes256_done(&ctx)
```

Finally, the header of the response APDU "`resp_APDU`" is composed with the value "0x9F" (Success) in the status word 1 and the value "0x10" in the status word 2 (number of bytes of data available to read in the data field). The variable "`LE`" contains the value of the size of the data field ("`SIZE_BLOCK_AES*N`") and the variable "`LEN`" contains the sum of the value "`LE`" and the size of a status word (2). The data field was filled in the last loop.

do_AES_NotEncrypt2 function

This function works in the same way than "`do_AES_Encrypt2`" except from this function does not execute the encryption code. The goal of this function is helpful to calculate the runtime of the encryption process, with subtracting of the runtime "`do_AES_Encrypt2`" function and "`do_AES_NotEncrypt2`" function.

The class of the command APDU expected is" 0x80" and the instruction is "0x09". The parameter 1 contains the number of blocks of 16 Bytes (as maximum 6).The parameter 2 is not required to have any value. The data field of the command APDU contains the data to copy in blocks of 16 Bytes (as maximum 96 Bytes). In the subsection "NotEncrypt2 class" appears the structure of the command APDU and response APDU.

The variable "`N`" contains the number of blocks to copy indicated in the parameter 1.

At the first, the array of unsigned integer "`key`" and "`buf`" are created with a length of 32 Bytes and the value of "`N`", respectively. The key used is stored in the array "`key`" and it is composed of the first 32 Bytes of the data field of the command APDU.

If the number of blocks to copy ("`N`") is higher than the maximum, the response APDU "`resp_APDU`" with the status word "6A 68" (referenced data not found) is returned and the command ends.

The structure of the main loop is the same than in "`do_AES_Encrypt2`", but now there are not any functions to encrypt. The loop consists in copy per blocks of 16 Bytes the data field of the command APDU "`com_APDU`" to "`buf`" and after that, copies the value of the array "`buf`" in the data field of the response APDU "`resp_APDU`". The loop starts to copy after the key.

```
for(i=0; i<N; i++){
        for(j=0; j<SIZE_BLOCK_AES; j++){
                buf[j] =
(*com_APDU).data_field[SIZE_KEY_AES+(i*SIZE_BLOCK_AES)+j];
        }
        for(j=0; j<SIZE_BLOCK_AES; j++){
                (*resp_APDU).data_field[(i*SIZE_BLOCK_AES)+j] = buf[j];
        }
}
```

Finally, the header of the response APDU "`resp_APDU`" is composed with the value "0x9F" (Success) in the status word 1 and the value "0x10" in the status word 2 (number of bytes of data available to read in the data field). The "LE" variable contains the value of the size of the data field ("SIZE_BLOCK_AES*N") and the "LEN" variable contains the sum of the value "LE" and the size of a status word (2). The data field was filled in the last loop.

test_timer1 function

This function measures the number of clock cycles necessary to copy the data available in the command APDU in a local array of unsigned integer. The timer/counter 1 is used to measure the clock cycles.

The class of the command APDU expected is" 0x80" and the instruction is "0x0A". The parameter 1 contains the length of the data field. The parameter 2 is not required to have any value. The data field of the command APDU contains the data to copy in the local array.

If the parameter 1 of the header does not contains a number higher than 0 and smaller than 15, the response APDU "resp" with the status word "6A 68" (referenced data not found) is returned and the function ends.

At the first, the timer 1 is initialized with the next values.

```
overflow = 0;
DDRB = 0xFF;
TCNT1L = 0x00;
TCNT1H = 0x00;
TCCR1A = 0;
TCCR1B = 1;
```

The variable "overflow" checks if an overflow happens between in sampling clock cycles. The register "DDRB" is set up to use all the pins on "PORTB" (8-bit bi-directional I/O port) for output. The registers "TCNT1L" (low byte) and "TCNT1H" (high byte) is set up to 0 to start value of timer/counter 1, because the timer 1 is a 16-bit timer and it has 2 registers to save the value. The register "TCCR1A" is set up to 0 to set the timer/counter 1 in timer mode. The register "TCCR1B" (timer/counter 0 control register) is set up to 1 to set the oscillator frequency.

The variables "cal_ref_timerL", "cal_ref_timerH", "cal_ref_timer2L" and "cal_ref_timer2H" contains the clock cycles between the main loop.

```
cal_ref_timerL = TCNT1L;
cal_ref_timerH = TCNT1H;
for(i=0; i<N; i++){
       buffer[i] = (*com_APDU).data_field[i];
}
cal_ref_timer2L = TCNT1L;
cal_ref_timer2H = TCNT1H;
```

It is necessary to check if the register "TIFR" (timer interrupt flag register) is set to avoid an overflow in the measure of the values of the timer/counter 1.

```
if((TIFR&0x04) == 0x04){
       overflow = 1;
       cal_ref_timer2L = 0x00;
       cal_ref_timer2H = 0x00;
}
```

The status word 1 of the response APDU "resp_APDU" is "SW1_SUCCESS" and the status word 2 is the amount of bytes in the data field, in this case 5. The data field consists of:

- First byte: value of "`cal_ref_timerH`".
- Second byte: value of "`cal_ref_timerL`".
- Third byte: value of "`overflow`". If this byte is set, an overflow happened in the measure of the previous values and they are incorrect.
- Fourth byte: value of "`cal_ref_timer2H`".
- Fifth byte: value of "`cal_ref_timer2L`".

do_AES_Encrypt_withTimers function

This function measures the number of clock cycles necessary to encrypt 16 Bytes of input plain text with a key 32 Bytes of a key returning 16 Bytes of cipher text as output and the values of the timers. The timer/counter 1 is used to measure the clock cycles.

The class of the command APDU expected is" 0x80" and the instruction is "0x0B". The parameter 1 and the parameter 2 are not required to have any value. The data field of the command APDU has to contain 16 Bytes of plain text to encrypt and the 32 Bytes of the key.

At the first, the array of unsigned integer "`key`" and "`buf`" are created with a length of 32 Bytes and 16 Bytes respectively. The key used in the encryption is stored in the array "`key`" and it is composed of the first 32 Bytes of the data field of the command APDU. The plain text to encrypt is stored in the array "`buf`" and it is composed of the next 16 Bytes (after the first 32 Bytes). In the subsection "Encrypt class" appears the structure of the command APDU.

The timer/counter 1 is initialized with the next values.

```
DDRB = 0xFF;
TCNT1L = 0x00;
TCNT1H = 0x00;
TCCR1A = 0;
TCCR1B = 1;
```

The register "`DDRB`" is set up to use all the pins on "`PORTB`" (8-bit bi-directional I/O port) for output. The registers "`TCNT1L`" (low byte) and "`TCNT1H`" (high byte) is set up to 0 to start value of timer/counter 1, because the timer 1 is a 16-bit timer and it has 2 registers to save the value. The register "`TCCR1A`" is set up to 0 to set the timer/counter 1 in timer mode. The register "`TCCR1B`" (timer/counter 0 control register) is set up to 5 to set the oscillator frequency.

In this function, the variable "`overflow`" does not exists cause it is stored in the byte eighteenth of the response APDU "`resp_APDU`". This byte checks if an overflow happens between in sampling clock cycles.

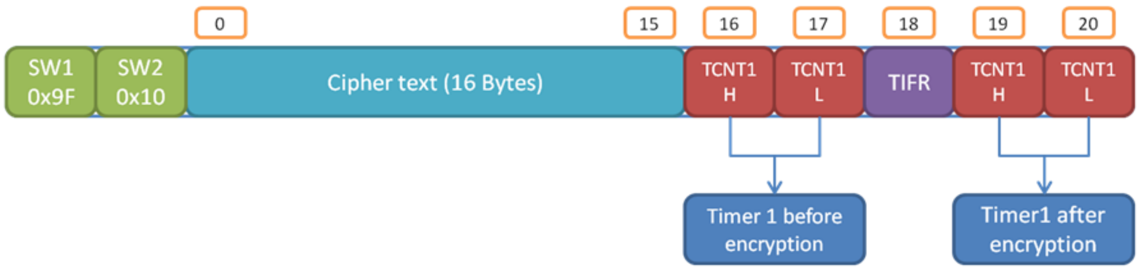
To reduce memory usage of the function, the variables to save the contents of the timer 1 before and after the encryption process are stored directly in the data field of the response APDU.

- 16th Byte: value of the register "`TCNT1H`" before encryption process.
- 17th Byte: value of the register "`TCNT1l`" before encryption process.
- 18th Byte: value of the register "`TIFR`" (overflow byte).
- 19th Byte: value of the register "`TCNT1H`" after encryption process.
- 20th Byte: value of the register "`TCNT1L`" after encryption process.

It is necessary to check if the register "TIFR" (timer interrupt flag register) is set to avoid an overflow in the measure of the values of the timer/counter 1

```
(*resp_APDU).data_field[18] = 0;
(*resp_APDU).data_field[17] = TCNT1L;
(*resp_APDU).data_field[SIZE_BLOCK_AES] = TCNT1H;
aes256_init(&ctx, key);
aes256_encrypt_ecb(&ctx, buf);
aes256_done(&ctx);
if((TIFR&0x04) != 0x04){
        (*resp_APDU).data_field[20] = TCNT1L;
        (*resp_APDU).data_field[19] = TCNT1H;
}else{
        (*resp_APDU).data_field[18] = 1;
        (*resp_APDU).data_field[20] = 0x00;
        (*resp_APDU).data_field[19] = 0x00;
}
```

Finally, in the data field of the response APDU is copied the array "buf", which contains the cipher text obtained. The header is composed with the value 0x9F (Success) in the status word 1 and the value 0x15 in the status word 2 (number of bytes of data available to read in the data field). The structure of the response APDU is showed below.



Structure of the response APDU "resp_APDU"

detailsApps function

This function returns a string encoded in UTF-8 in the data field of the response APDU "resp_APDU". This string contents a short explanation of the features developed in the smart card.

The class of the command APDU expected is" 0x80" and the instruction is "0x0C". The parameter 1 and the parameter 2 are not required to have any value. The data field of the command APDU contains the description of the smart card.

In the array of unsigned char "name" appears the description of the smart card that it is copied in the next loop to the data field of "resp_APDU".

```
for(i=0; i<29; i++){
        (*resp_APDU).data_field[i] = name[i];
}
```

Finally, the header of the response APDU "resp_APDU" is composed with the value "0x9F" (Success) in the status word 1 and the value "0x1D" in the status word 2 (number of bytes of data available to read in the data field).

do_AES_NotEncrypt function

This command works in the same way than "do_AES_Encrypt" except from this function does not execute the encryption code. The goal of this command is helpful to calculate the runtime of the encryption process, with subtracting of the runtime "`do_AES_Encrypt`" command and "`do_AES_NotEncrypt`" command.

The class of the command APDU expected is" 0x80" and the instruction is "0x0D". The parameter 1 and the parameter 2 are not required to have any value. The data field of the command APDU contains the data to copy. In the subsection "NotEncrypt class" appears the structure of the command APDU and response APDU.

At the first, the array of unsigned integer "`key`" and "`buf`" are created with a length of 32 Bytes and 16 Bytes respectively. The key is stored in the array "`key`" and it is composed of the first 32 Bytes of the data field of the command APDU. The plain text to copy is stored in the array "`buf`" and it is composed of the next 16 Bytes (after the first 32 Bytes).

The structure of the main loop is the same than in "`do_AES_Encrypt`", but now there are not any functions to encrypt. The loop consists in copy the data field of the command APDU "`com_APDU`" to "`buf`" and after that, copies the value of the array "`buf`" in the data field of the response APDU "`resp_APDU`". The loop starts to copy after the key.

```
for(i=0; i<SIZE_KEY_AES; i++){
       key[i] = (*com_APDU).data_field[i];
}
for(i=0; i<SIZE_BLOCK_AES; i++){
       buf[i] = (*com_APDU).data_field[i+SIZE_KEY_AES];
}
for(i=0; i<SIZE_BLOCK_AES; i++){
       (*resp_APDU).data_field[i] = buf[i];
}
```

Finally, the header of the response APDU "`resp_APDU`" is composed with the value "0x9F" (Success) in the status word 1 and the value "0x10" in the status word 2 (number of bytes of data available to read in the data field).

command_Handler function

This is the main function of the source file "functions_smartcard.c". This function selects the code of the command (implemented previously) to execute depending of the class and instruction of the command APDU.

The node address (NAD) and the protocol control byte (PCB) of the response APDU "`resp_APDU`" has the same value than the command APDU "`com_APDU`".

Depending of the block structure of the command APDU, the system blocks (S blocks) are managed in a different ways than the information blocks (I Blocks).

There are a switch to select the command to execute, and if there is an error selecting the application, the respond APDU is made up with the status word "6E 00" (Incorrect application, CLA parameter of a command). If there is an error selecting the command, the respond APDU is made up with the status word "6E 00" (command not allowed, invalid instruction byte). Otherwise, the command is selected successfully.

main.c

In this file appears the source code to execute every time the smart card is connected to a smart card reader.

At first, the structure "command_APDU" called "rec_APDU" and the structure "response_APDU" called "res_APDU" are initialized. The next loop waits sending an ATR (Answer to Reset) at least 400 cycles.

The main endless loop waits to receive a command APDU with the function "receive_APDU". When a command APDU is received, the return code of the last function has to be checked to prevent a Checksum error.

If there is an error, the response APDU "res_APDU" with the status word "67 01" (incorrect length or address range error) is returned.

If the receipt of the command APDU is successfully, the function "command_Handler" (functions_smartcard.c) is executed with the arguments as pointer "rec_APDU" and "res_APDU".

The response APDU "res_APDU" is sent out to the smart card and the variables of the structure "res_APDU" are initialized to default values to repeat another time the endless loop.

External dependencies

The files that make up this section are mostly header files included in the previously header files and source files. These external dependencies are provided by the modules available from AVR libc, except the library "libT1_Comm_Lib.a" that is added directly to the project and provides the functions related in the header file "libT1_Comm_Lib.h".

Results and conclusion

Runtime measurements

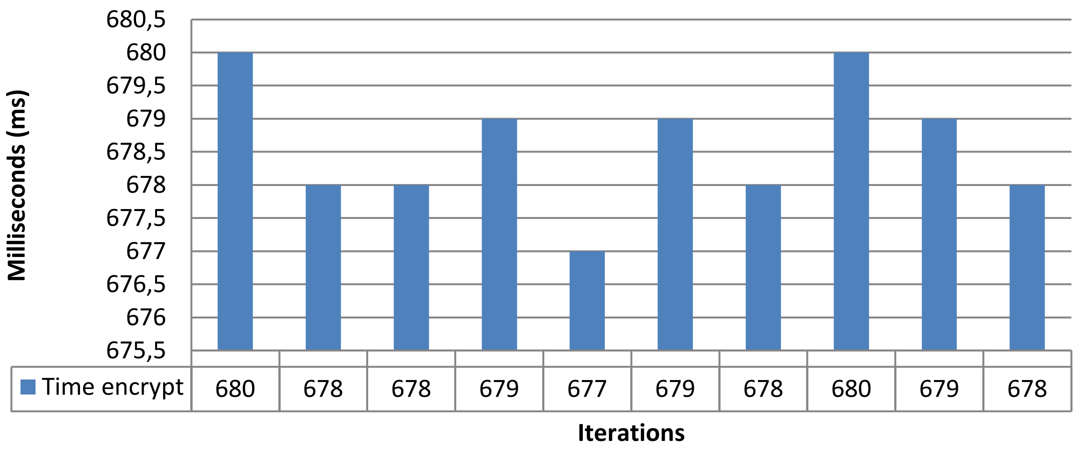
As a result of the code written, several tests have been done in the code and the result has been successful. The code to encrypt and to decrypt works fine and below appears the graphics of the runtime for each encrypt class (Encrypt class and Encrypt2 class).

It is worth mentioning than in the Encrypt class, when the command APDU is sent, selects the "`do_AES_Encrypt`" function (class 128, instruction 2). In the Encrypt2 class, when the command APDU is sent, selects the "`do_AES_Encrypt2`" function (class 128, instruction 8). The difference between these two functions is explained previously in the section AES encryption and decryption.

The stats of the runtime of Encrypt class and Encrypt2 class are detailed below. The next stats have a margin of error as maximum of 3 or 4 milliseconds in 10 iterations, so the results obtained are right.

Example runtime Encrypt class with 32 Bytes as plain text

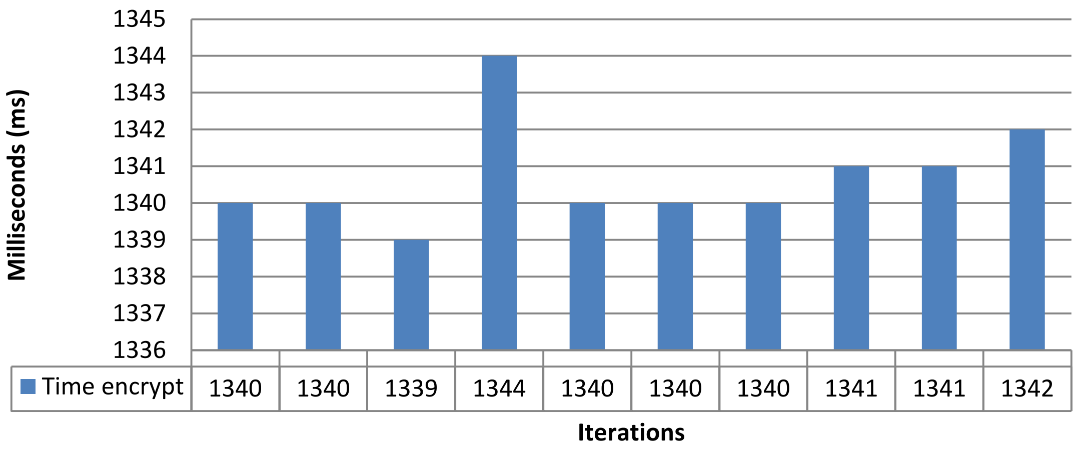
Average of 10 iterations: 678.6 milliseconds



🟦 Time encrypt	680	678	678	679	677	679	678	680	679	678

Example runtime Encrypt class with 64 Bytes as plain text

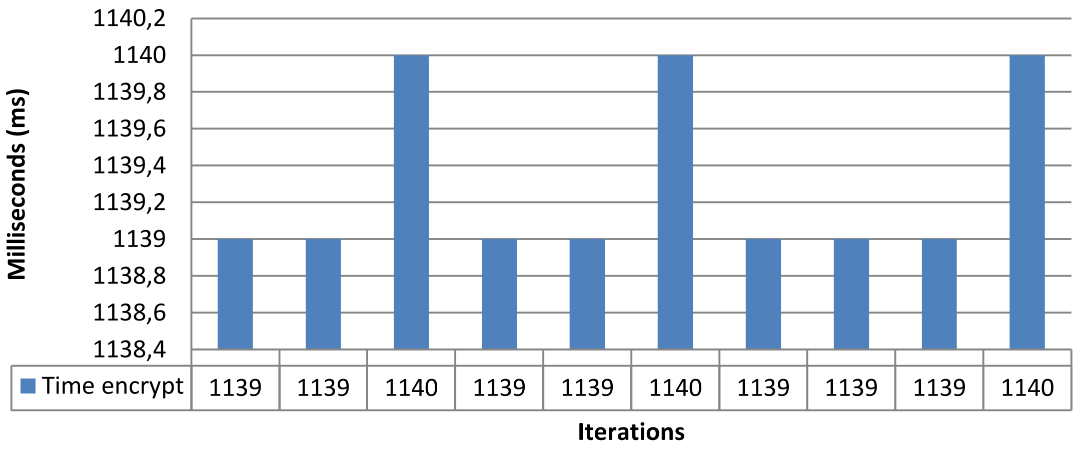
Average of 10 iterations: 1340.7 milliseconds



■ Time encrypt	1340	1340	1339	1344	1340	1340	1340	1341	1341	1342

Example runtime Encrypt2 class with 64 Bytes as plain text

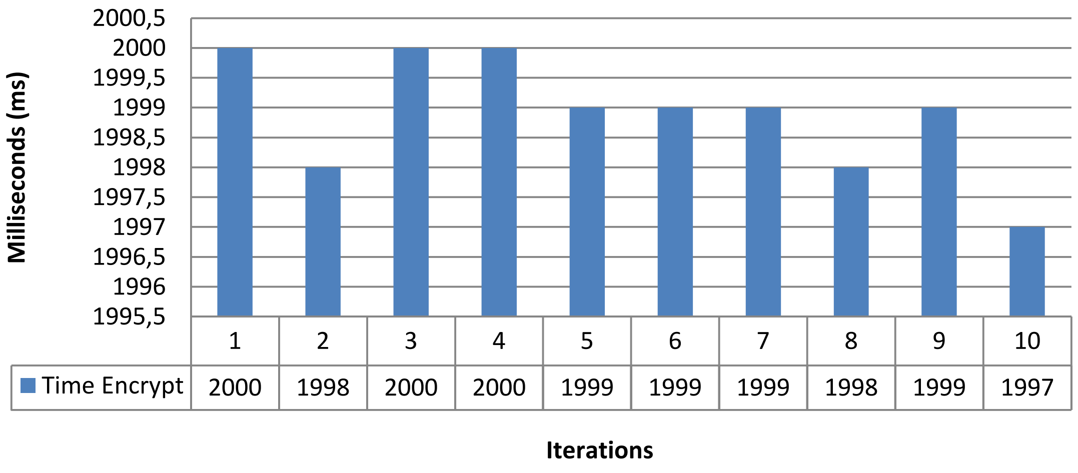
Average of 10 iterations: 1139.3 milliseconds



■ Time encrypt	1139	1139	1140	1139	1139	1140	1139	1139	1139	1140

Example runtime Encrypt class with 96 Bytes as plain text

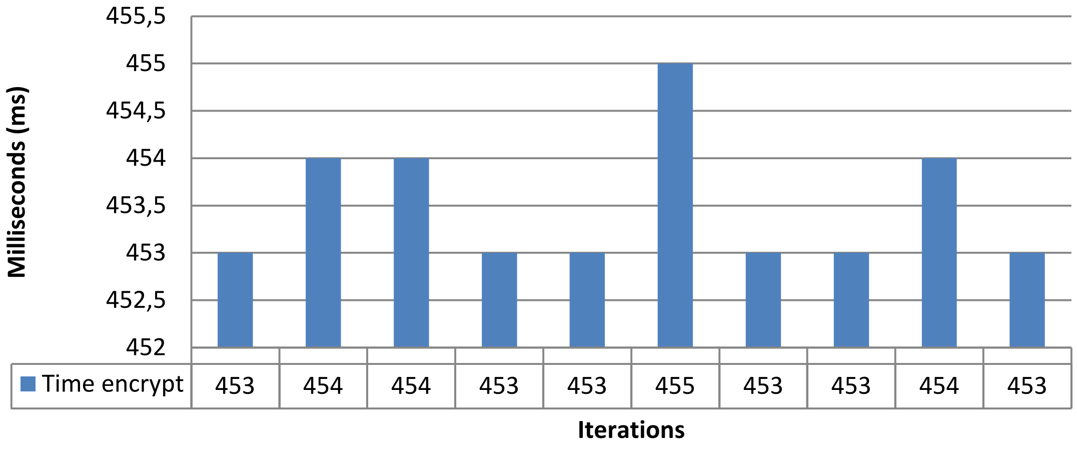
Average of 10 iterations: 1998.9 milliseconds



	1	2	3	4	5	6	7	8	9	10
■ Time Encrypt	2000	1998	2000	2000	1999	1999	1999	1998	1999	1997

Example runtime Encrypt2 class with 96 Bytes as plain text

Average of 10 iterations: 453.5 milliseconds



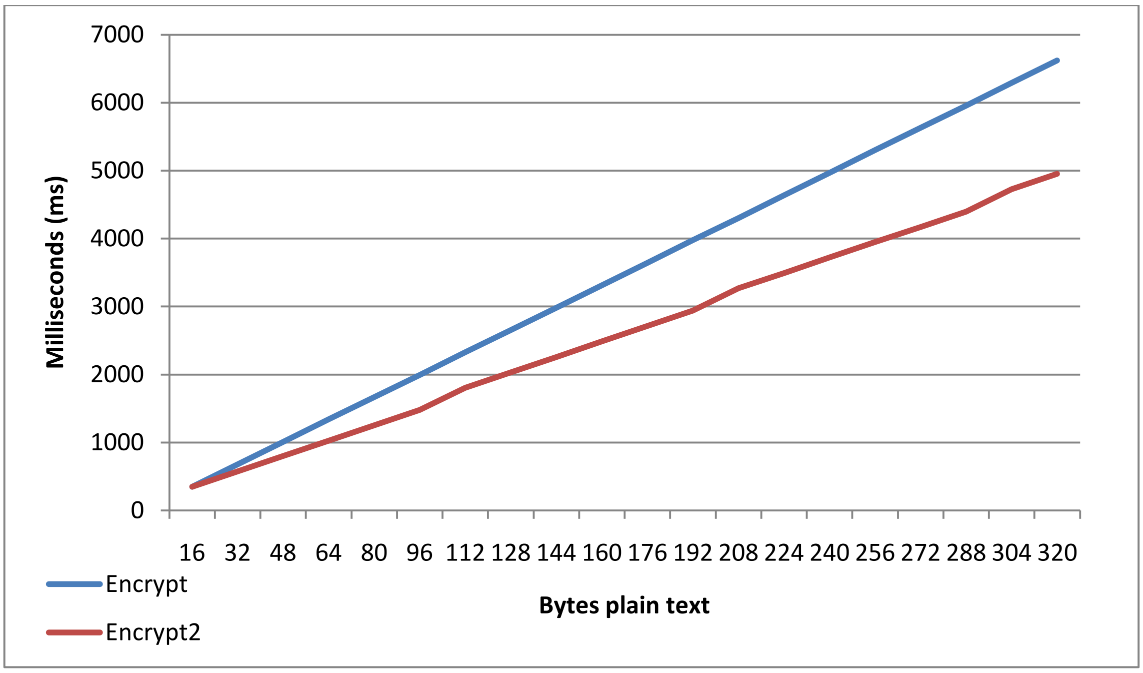
🟦 Time encrypt	453	454	454	453	453	455	453	453	454	453

Runtime of Encrypt class and Encrypt2 class

The below statistics are made with real data until 320 Bytes of plain text. Each value is calculated from 10 iterations of runtime. The stat of Encrypt class has a constant slop, because the data field of the command APDU always has the same size, but the larger the plain text to encrypt, the greater the number of commands APDU sent. However, Encrypt2 class does not work in the same way than Encrypt class, because the size of the data field of a command APDU sent can change.

The size of the data field does not influence a lot in the runtime of a command APDU, but sending more than one command APDU to the smart card influences in the total runtime of encryption. That is the reason because appears some peaks in the graph of Encrypt2, and they appear in the transition of send another extra command APDU to continue encrypting the plain text. The peaks appear between 96 and 112 Bytes, 192 and 208 Bytes, 288 and 304 Bytes.

Besides, the difference between two classes increases with increase plain text, and it is a good improvement of Encrypt2 class. With 1024 Bytes of plain text, the runtime difference between both classes is 5515,5 milliseconds.



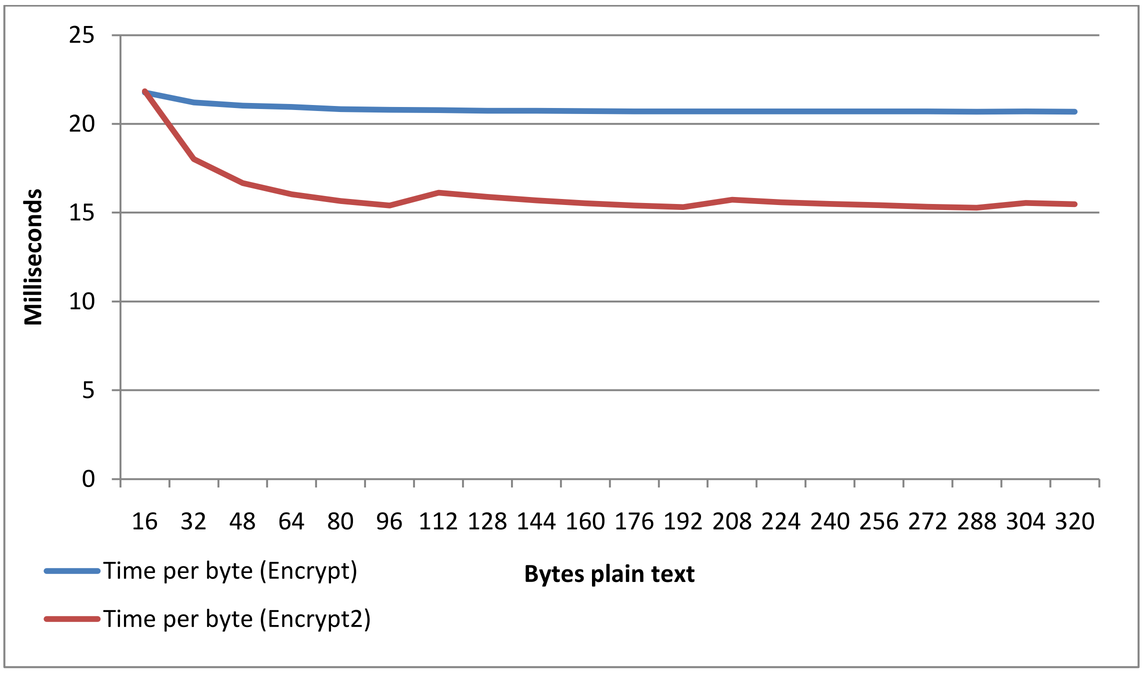
Bytes plain text	Encrypt (ms)	Encrypt2 (ms)
16	348,2	349,4
32	678,6	576,3
48	1008,9	800,22
64	1340,7	1025,9
80	1666,2	1253
96	1995,2	1478,8
112	2326,5	1806,6
128	2654,3	2032,8
144	2985	2258,4
160	3314,3	2485,9
176	3643,6	2712,2
192	3976,2	2939,4
208	4304,6	3269,9
224	4636,8	3491,5
240	4967,1	3720,4
256	5299,2	3946,9
272	5629,1	4172,1
288	5958,6	4397,9
304	6291,2	4724,3
320	6620,5	4952,5

Encrypt2 class runtime and Encrypt class runtime per data field load

The below Statistics are made with real data until 320 Bytes of plain text. Each value is calculated from 10 iterations of runtime. The statistic represents the runtime of encrypt 1 Byte depending the data field load of the command APDU.

The graph of time per byte in Encrypt class remains more or less lineal, because the runtime to send a command APDU is always the same because the length of data field of this command APDU is always the same.

However, the changes of size of the data field in the Encrypt2 class force to reduce the runtime in the first command APDU sent to encrypt the data, but after that, the graph remains more or less lineal. Between sending an extra command APDU appears peaks, in the same way than in "Runtime of Encrypt class and Encrypt2 class".



Bytes plain text	Time per byte (Encrypt)	Time per byte (Encrypt2)
16	21,7625	21,8375
32	21,20625	18,009375
48	21,01875	16,67125
64	20,9484375	16,0296875
80	20,8275	15,6625
96	20,78333333	15,40416667
112	20,77232143	16,13035714
128	20,73671875	15,88125
144	20,72916667	15,68333333
160	20,714375	15,536875
176	20,70227273	15,41022727
192	20,709375	15,309375
208	20,69519231	15,72067308
224	20,7	15,58705357
240	20,69625	15,50166667
256	20,7	15,41757813
272	20,69522059	15,33860294
288	20,68958333	15,27048611

304	20,69473684	15,54046053
320	20,6890625	15,4765625

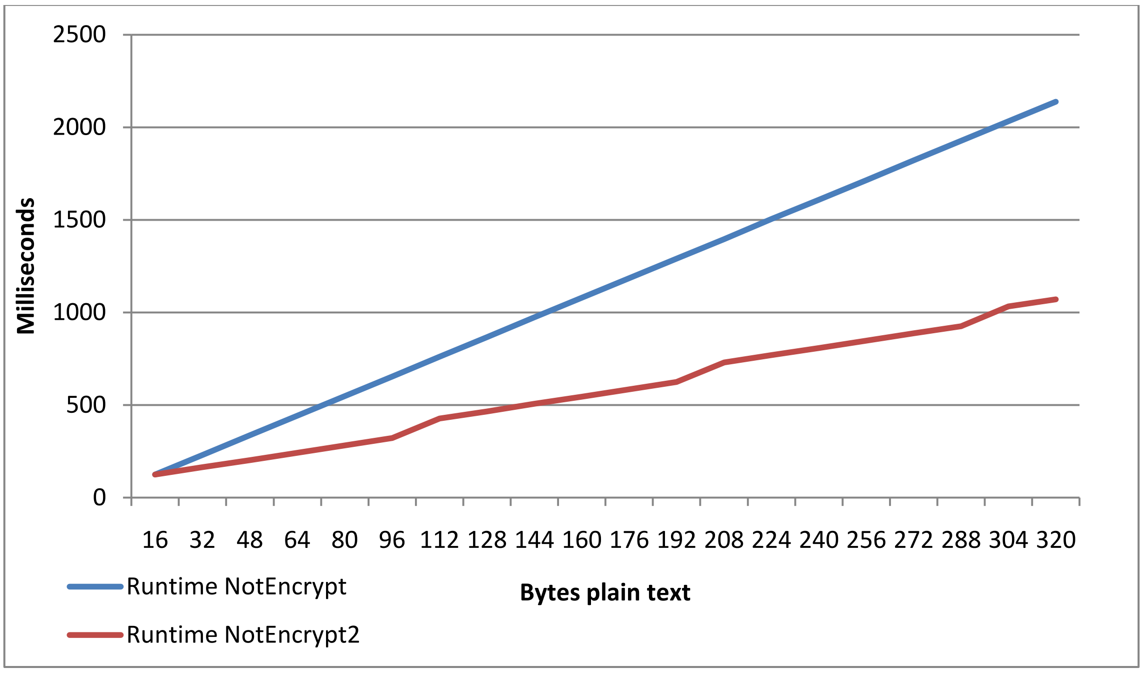
The average of Encrypt class per byte of plain text sent is 20,82 milliseconds. The average of Encrypt2 class per byte of plain text sent is 16,015 milliseconds.

Runtime NotEncrypt class and NotEncrypt2 class

This statistic represents the runtime of a command APDU that executes the command "`test_command_withData2`" (NotEncrypt2 class) and "`do_AES_NotEncrypt`" (NotEncrypt class).

The command "`test_command_withData2`" works on the same way than the command "`do_AES_Encrypt2`" except that the encryption code is not executed. The command "`do_AES_NotEncrypt`" works also on the same way than the command "`do_AES_Encrypt`" except that the encryption code is not executed.

This graph has the same shape than "Runtime Encrypt and Encrypt2". The graph of NotEncrypt class remains lineal against the NotEncrypt2 class. In this case, the peaks of NotEncrypt2 are higher than in the other statistic, because now the encryption code does not influence in the final runtime.



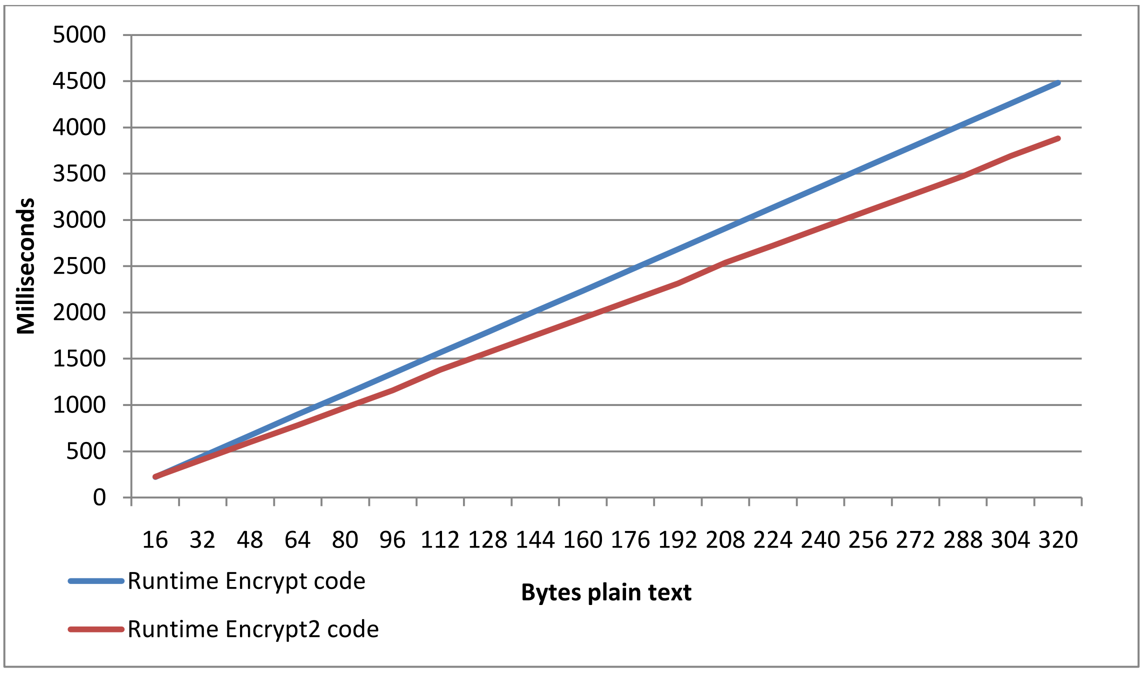
Bytes plain text	Runtime Not Encrypt	Runtime Not Encrypt2
16	124,9984	125,4
32	230,8993	164,3
48	336,7997	203,22
64	442,7989	242,3
80	548,5998	282,2
96	654,5998	321,2

112	760,3997	426,9
128	866,2992	465,6
144	972,8999	506,1
160	1078,4999	544,9
176	1184,4998	584,1
192	1291,0995	623,55
208	1395,8996	729,6
224	1503,9968	769
240	1608,6983	808,4
256	1714,4994	847,7
272	1820,5998	886,9
288	1926,1999	925,9
304	2032,7999	1031,9
320	2137,9999	1071,3

Runtime of encryption code (Encrypt2 class)

The next statistic represents the runtime of encryption code, obtained from subtracting the runtime of the commands “`do_AES_Encrypt2`” and “`test_command_withData2`”.

The fact for which there is a difference between the runtime of Encrypt code and Encrypt2 code is because the structure “`aes256_context`” in the command “`do_AES_Encrypt2`” (Encrypt2 class) is initialized and released one time for more than one block to encrypt. Nevertheless, in the command “`do_AES_Encrypt`” this structure is initialized and released with a command APDU, in other words, with every block to encrypt.



The average of the encryption code per byte is 12,14 milliseconds with the command “`do_AES_Encrypt2`”.

Bytes plain text	Runtime Encrypt code	Runtime Encrypt2 code
16	223,2016	224
32	447,7007	412
48	672,1003	597
64	897,9011	783,6
80	1117,6002	970,8
96	1340,6002	1157,6
112	1566,1003	1379,7
128	1788,0008	1567,2
144	2012,1001	1752,3
160	2235,8001	1941
176	2459,1002	2128,1
192	2685,1005	2315,85
208	2908,7004	2540,3
224	3132,8032	2722,5
240	3358,4017	2912
256	3584,7006	3099,2
272	3808,5002	3285,2
288	4032,4001	3472
304	4258,4001	3692,4
320	4482,5001	3881,2

Conclusions

In agreement with the values of the statistics, the difference between the runtime of the encryption code and sending the command APDU is 4 milliseconds, so the most runtime of sending a command APDU to execute the command "`do_AES_Encrypt2`" is spent to execute the encryption code.

The runtime of the encryption code is faster in the Encrypt2 class because in the command "`do_AES_Encrypt2`", the structure is "`aes256_context`" initialized and released only one time to encrypt more than 16 Bytes. However, in the Encrypt class this structure is initialized and released once to encrypt 16 Bytes each time. The difference of runtime of the encryption code for 1024 Bytes of plain text is 1981.8002 milliseconds.

The most difference of runtime between Encrypt class and Encrypt2 class is when these classes send the data (command APDU) and receive the data (response APDU). The NotEncrypt and NotEncrypt2 classes have been used to calculate the runtime of send and receive the data without execute any encryption code in the smart card. The difference is quite significant because sometimes the NotEncrypt2 class send 96 Bytes as plain text, the maximum available capacity of the command APDU. With 1024 Bytes of plain text, the difference between NotEncrypt and NotEncrypt2 classes reaches until 3533.6998 milliseconds.

The limit of 96 Bytes to send or receive with the smart card is not the maximum, due theoretically; the data field of a command APDU has 256 Bytes as maximum length. In this case, if sent more than 96 Bytes, the smart card return an error code, due an overflow with the memory thus is another limit to not reduce the runtime of Encrypt2 class.

The smart card can be used to encrypt small pieces of plain text because the runtime of encryption is more or less acceptable, until 1024 Bytes. To encrypt higher amounts of plain text, these functions are not recommended because the smart card will spent more than 15 seconds to do the encryption.

The implementation of AES-32 integrated in the smart card uses the simplest mode of encryption, the electronic codebook (ECB). The reason of usage of this mode of encryption is to accelerate the runtime of the encryption code because it is oriented to 8-bit microcontrollers. This mode is not the most secure of all of the others, but if it used to a small amount of plain text, it should works fine in a security perspective.

From the above causes, it is recommended to use the commands to encrypt or to decrypt of the smart card to sizes of plain text smaller than 1024 Bytes, for example, the text of an e-mail. A new idea to use for this smart card is to encrypt or decrypt messages shared with some users via instant messaging applications. That requires a modification of the applications to be able to use the smart card to encrypt or decrypt the text and it can be developed like a plug-in.

The code developed in Java is directed to use in a console mode, but it can be easily modified to add a graphic user interface. This graphic user interface will allow to the user selects the file to encrypt or decrypt and the key required, and selects the destination folder where save the output generated.

To recreate a possible usage scenario of the smart card, the key would be stored into the personal smart card of the user. With that smart card, the user would be able to encrypt or decrypt whatever file requested, in an easily way. The problem would be appear if the smart card is stolen from an attacker, because the key is stored into the smart card, but other security solutions can be integrated to this scenario.

Bibliography

- Smart card article in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/Smart_card consulted in November 2009.
- Contactless smart card article in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/Contactless_smart_card consulted in November 2009.
- RANKL, Wolfgang. Smart card applications: design models for using and programming smart cards, Wiley, June 2007. Chapters consulted:
    - Chapter 2.3.3 Transmission protocols (p. 24).
    - Chapter 2.3.3.1 T=0 transmission protocol for contact smart cards (p. 25).
    - Chapter 2.3.3.2 T=1 Transmission protocol for contact smart cards (p. 25).
- RANKL, Wolfgang; EFFING, Wolfgang. Smart Card Handbook, Wiley, January 2004. Chapters consulted:
    - 6.5.2 Structure of the response APDU (p. 424).
    - 6.4.3 The T=1 transmission protocol (p. 409).
    - 11.4.1 PC/SC (p. 667).
    - 11.4.4 MUSCLE (p. 672).
    - 6.2 Answer To Reset (ATR) (p. 377).
- PC/SC article in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/PC/SC consulted in November 2009.
- Java Card article in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/Java_Card consulted in November 2009.
- Java Card Technology Overview article in Sun Developer Network (SDN). http://java.sun.com/javacard/overview.jsp consulted in November 2009.
- Module scard article in pyscard (sourceforge). http://pyscard.sourceforge.net/epydoc/smartcard.scard.scard-module.html consulted in November 2009.
- Direct application selection article in CardWerk (Smarter Card Solutions). http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4_9_application-independent_card_services.aspx#ISO7816-4_9_3_2 consulted in November 2009.
- Triple DES article in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/Triple_DES consulted in November 2009.
- Data Encryption Standard in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/Data_Encryption_Standard consulted in November 2009.
- Answer to reset article in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/Answer_to_reset consulted in November 2009.
- Advanced Encryption Standard article in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard consulted in December 2009.
- Electronic codebook (ECB) section of the Block cipher modes of operation article in Wikipedia, publication in Internet. http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Electronic_codebook_.28ECB.29 consulted in December 2009.

- STALLINGS, William. Cryptography and Network Security: Principles and Practice, Prentice Hall, January 2010 (5 edition). Chapter 6.2 Electronic code book consulted (p. 198).
- AVRFREAKS. Timer/Counter Basics (Design note #024), http://www.avrfreaks.net/modules/FreaksFiles/files/388/DN_024.pdf consulted in November 2009.
- ATMEL. 8-Bit AVR Microcontroller with 8K bytes In-System Programmable Flash AT90S8515, http://www.atmel.com/atmel/acrobat/doc1142.pdf consulted in November 2009.
- ATMEL. 8-Bit AVR Microcontroller with 16K bytes In-System Programmable Flash ATmega163 and ATmega163L, http://www.atmel.com/dyn/resources/prod_documents/doc1142.pdf consulted in November 2009.
- PC-Link Readers article in Gemalto web. http://www.gemalto.com/products/pc_link_readers/ consulted in October 2009.
- Dynamite Programmer +Plus article in Duolabs web. http://www.duolabs.com/dynamite.html consulted in October 2009.
- DUOLABS. Cas Interface 3 User's Guide, May 2005 published in Internet. http://www.qboxsvn.com/duolabs/ManualeCas3_EN.pdf consulted in October 2009.
- AVR Studio 4 article in Atmel web. http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=2725 consulted in November 2009.
- ATMEL. AVR Studio 3.5 User guide 2001. http://www.atmel.com/dyn/resources/prod_documents/doc1663.pdf consulted in January 2010.
- Introduction article in AVR Libc web site. http://www.cs.mun.ca/~paul/cs4723/material/atmel/avr-libc-user-manual-1.6.5/ consulted in November 2009.
- Library reference in AVR Libc web site. http://www.cs.mun.ca/~paul/cs4723/material/atmel/avr-libc-user-manual-1.6.5/modules.html consulted in November 2009.
- Eclipse (software) article in Wikipedia. http://en.wikipedia.org/wiki/Eclipse_(software) consulted in November 2009.
- Eclipse Java development tools (JDT) Overview article in Eclipse website. http://www.eclipse.org/jdt/overview.php consulted in November 2009.
- Padding (cryptography) article in Wikipedia. http://en.wikipedia.org/wiki/Padding_(cryptography) consulted in January 2010.
- Java TM Smart Card I/O API in Package javax.smartcardio. http://java.sun.com/javase/6/docs/jre/api/security/smartcardio/spec/ consulted in December 2009.
- Smart Card Authentication article in MSDN Library. http://msdn.microsoft.com/en-us/library/aa380142(v=VS.85).aspx consulted in November 2009.

- Application Example and Algorithms article in Atmel website. http://www.atmel.com/dyn/products/app_notes.asp?part_id=2027 consulted in November 2009.
- GlobalPlatform article in Wikipedia. http://en.wikipedia.org/wiki/GlobalPlatform consulted in November 2009.
- An Introduction to Java Card Technology article in Sun Developer Network (SDN). http://java.sun.com/javacard/reference/techart/javacard1/ consulted in December 2009.
- Introduction to cryptography, Part 2: Symmetric cryptography article in IBM deveolperWorks. http://www.ibm.com/developerworks/library/s-crypt02.html

Appendixes

Interface in Java with the smart card

Encrypt class

```
import javax.smartcardio.*;


/** Class to encrypt data in the smart card (without timers)
 *  Input files:
 *      (1) "input": this file contains the data to encrypt,
 *      (2) "key": this file contains 32 bytes of data that contains the key.
 *  Output files:
 *      (1) "output-encrypted": this file contains the data encrypted.
 */
public class Encrypt extends Functions{

    public static void main(String args[]){
       long time = 0;

       Card card = connectCard();
       CardChannel channel = establishChannel(card);

         byte[] key = readFile("C:/TMP/key");
         byte[] plainText = readFile("C:/TMP/input");

         byte[] dataIn, dataOut, plainTextBuf, dataOutBuf;
         dataOut = null;
         plainTextBuf = new byte[SIZE_BLOCK_AES];
         dataOutBuf = new byte[SIZE_BLOCK_AES];

         plainText = addPadding(plainText);
         dataOut = new byte[plainText.length];
         System.out.println("Length data to encrypt: "+plainText.length);
         try{
             if(plainText.length>SIZE_BLOCK_AES){
              int N = plainText.length / SIZE_BLOCK_AES;
                 for(int i=0; i<N; i++){
                     // take the segment of plainText to encrypt
                     for(int j=0; j<SIZE_BLOCK_AES; j++){
                         plainTextBuf[j] = plainText[j+(i*SIZE_BLOCK_AES)];
                     }
                     dataIn = concatenateArrayByte(key, plainTextBuf);
                     dataOutBuf = sendAPDUwithData(CLA, INS_AES_ENCRYPT, 0, 0,
dataIn, channel);
                     time += getTimeExecution();
                     for(int j=0; j<SIZE_BLOCK_AES; j++){
                         dataOut[j+(i*SIZE_BLOCK_AES)] = dataOutBuf[j];
                     }
                 }
             }else{
                 dataIn = concatenateArrayByte(key, plainText);
                 dataOut = sendAPDUwithData(CLA, INS_AES_ENCRYPT, 0, 0, dataIn,
channel);
                 time = getTimeExecution();
             }
         }catch(NullPointerException ex3){
             System.out.println(ex3.getMessage());
             System.exit(-1);
         }
         writeFile("C:/TMP/output-encrypted", dataOut);
         disconnectCard(card);
         System.out.println("Encryption duration: "+time);
         System.out.println("Encryption succesful!");
    }
}
/** End Encrypt.java */
```

Decrypt class

```
import javax.smartcardio.*;

/** Class to decrypt data in the smart card
 *  Input files:
 *      (1) "output-encrypted": this file contains the data to decrypt,
 *      (2) "key": this file contains 32 bytes of data that contains the key.
 *  Output files:
 *      (1) "output-cipherText": this file contains the plain text decrypted.
 */
public class Decrypt extends Functions{

      public static void main(String args[]){
            long time = 0;
            Card card = connectCard();
            CardChannel channel = establishChannel(card);

            byte[] key = readFile("C:/TMP/key");
            byte[] cipherText = readFile("C:/TMP/output-encrypted");

            byte[] dataIn, dataOut, cipherTextBuf, dataOutBuf;
            dataOut = new byte[cipherText.length];
            cipherTextBuf = new byte[SIZE_BLOCK_AES];
            dataOutBuf = new byte[SIZE_BLOCK_AES];

            System.out.println("Length data to decrypt:
"+cipherText.length);
            if(cipherText.length > SIZE_BLOCK_AES){
                  int N = cipherText.length/SIZE_BLOCK_AES;
                  for(int i=0; i<N; i++){
                        for(int j=0; j<SIZE_BLOCK_AES; j++){
                              cipherTextBuf[j] =
cipherText[j+(i*SIZE_BLOCK_AES)];
                        }
                        dataIn = concatenateArrayByte(key, cipherTextBuf);
                        // Management exception NullPointerException
                        dataOutBuf = sendAPDUwithData(CLA, INS_AES_DECRYPT,
0, 0, dataIn, channel);
                        time += getTimeExecution();
                        for(int j=0; j<SIZE_BLOCK_AES; j++){
                              dataOut[j+(i*SIZE_BLOCK_AES)] =
dataOutBuf[j];
                        }
                  }
            }else{
                  dataIn = concatenateArrayByte(key, cipherText);
                  dataOut = sendAPDUwithData(CLA, INS_AES_DECRYPT, 0, 0,
dataIn, channel);
                  time = getTimeExecution();
            }

            dataOut = deletePadding(dataOut);
            Functions. WriteFile("C:/TMP/output-plainText", dataOut);
            disconnectCard(card);
            System.out.println("Decryption duration: "+time);
            System.out.println("Decryption succesful!");
      }
}/** End Decrypt.java */
```

Encrypt2 class

```
import javax.smartcardio.*;
```

```
/** Class to encrypt data in the smart card with a improved way.
 *    In one APDU can be send more data (higher SIZE_BLOCK_AESB) to encrypt
 *  Input files:
 *      (1) "input": this file contains the data to encrypt,
 *      (2) "key": this file contains SIZE_KEY_AES bytes of data that contains
the key.
 *  Output files:
 *      (1) "output-encrypted": this file contains the data encrypted.
 */
public class Encrypt2 extends Functions{

       public static void main(String args[]){
              long time = 0;
//            boolean ov32 = false;

              Card card = connectCard();
       CardChannel channel = establishChannel(card);

         byte[] key = readFile("C:/TMP/key");
         byte[] plainText = readFile("C:/TMP/input");

         byte[] dataIn, dataOut, plainTextBuf, dataOutBuf;
//         byte[] plainTextBuf2 = new byte[16];
         dataOut = null;
         plainTextBuf = null;
         dataOutBuf = new byte[SIZE_BLOCK_AES];

         plainText = addPadding(plainText);
         System.out.println("Length data to encrypt: "+plainText.length);

//         if(plainText.length%MAX_BYTES_AES == 32){
//            ov32 = true;
//            for(int i=0; i<SIZE_BLOCK_AES; i++){
//                   plainTextBuf2[i] = plainText[i+(plainText.length-
SIZE_BLOCK_AES)];
//            }
//            plainTextBuf = new byte[plainText.length-SIZE_BLOCK_AES];
//            for(int i=0; i<plainTextBuf.length; i++){
//                   plainTextBuf[i] = plainText[i];
//            }
//            plainText = plainTextBuf;
//         }

         int N = plainText.length / (SIZE_BLOCK_AES*MAX_N_BLOCKS_AES);
//         if(ov32){
//            dataOut = new byte[plainText.length+SIZE_BLOCK_AES];
//         }else{
              dataOut = new byte[plainText.length];
//         }
         if(N>0){
              if(plainText.length % (SIZE_BLOCK_AES*MAX_N_BLOCKS_AES) != 0){
                     N++;
              }

              for(int i=0; i<N; i++){
                     int a = i*SIZE_BLOCK_AES*MAX_N_BLOCKS_AES;
                     int b = (i+1)*SIZE_BLOCK_AES*MAX_N_BLOCKS_AES;
                     if(plainText.length < b){
                            plainTextBuf = new byte[plainText.length - a];
                            for(int j=0; j<(plainText.length - a); j++){
                                   plainTextBuf[j] = plainText[j+a];
                            }
                     }else{
                            plainTextBuf = new byte[MAX_BYTES_AES];
                            for(int j=0; j<MAX_BYTES_AES; j++){
                                   plainTextBuf[j] =
plainText[j+(i*MAX_BYTES_AES)];
                                }
                        }

                        dataIn = concatenateArrayByte(key, plainTextBuf);
                        int P1 = (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES;
                        try{
                                dataOutBuf = sendAPDUwithData(CLA,
INS_AES_ENCRYPT_V2, P1, 0, dataIn, channel);
                                time += getTimeExecution();
                        }catch(NullPointerException ex1){
                                System.out.println(ex1.getMessage());
                                System.exit(-1);
                        }
                        for(int j=0; j<dataOutBuf.length; j++){
                                dataOut[j+(i*(SIZE_BLOCK_AES*MAX_N_BLOCKS_AES))] =
dataOutBuf[j];
                        }
                }
        }else{
                dataIn = concatenateArrayByte(key, plainText);
                int P1 = (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES;
                // Revise Exception NullPointerException
                dataOutBuf = sendAPDUwithData(CLA, INS_AES_ENCRYPT_V2, P1, 0,
dataIn, channel);
                time = getTimeExecution();
                for(int i=0; i<dataOutBuf.length; i++){
                        dataOut[i] = dataOutBuf[i];
                }
        }
//        if(ov32){
//              dataIn = concatenateArrayByte(key, plainTextBuf2,
plainTextBuf2.length);
//              dataOutBuf = sendAPDUwithData(CLA, INS_AES_ENCRYPT,
dataIn.length, 0, dataIn, channel);
//              time += getTimeExecution();
//              for(int i=0; i<SIZE_BLOCK_AES; i++){
//                      dataOut[i+(plainText.length)] = dataOutBuf[i];
//              }
//        }
        writeFile("C:/TMP/output-encrypted", dataOut);
        disconnectCard(card);
        System.out.println("Encryption2 duration: "+time);
        System.out.println("Encryption2 succesful!");
       }

}/** End Encrypt2.java */
```

Decrypt2 class

```
import javax.smartcardio.*;

/** Class to decrypt data in the smart card with a improved way.
 *     In one APDU can be send more data (higher SIZE_BLOCK_AESB) to decrypt
 *  Input files:
 *      (1) "output-encrypted": this file contains the data to decrypt,
 *      (2) "key": this file contains SIZE_KEY_AES bytes of data that contains
the key.
 *  Output files:
 *      (1) "output-plainText": this file contains the data encrypted.
 */
public class Decrypt2 extends Functions{

        public static void main(String args[]){
                long time = 0;
//              boolean ov32 = false;
```

```
		Card card = connectCard();
		CardChannel channel = establishChannel(card);

		byte[] key = readFile("C:/TMP/key");
		byte[] cipherText = readFile("C:/TMP/output-encrypted");

		byte[] dataIn, dataOut, cipherTextBuf, dataOutBuf;
//		byte[] cipherTextBuf2;
//		cipherTextBuf2 = new byte[16];
		dataOut = new byte[cipherText.length];
		dataOutBuf = new byte[SIZE_BLOCK_AES];

		System.out.println("Length data to decrypt:
"+cipherText.length);

//		if(cipherText.length%MAX_BYTES_AES == 32){
//			ov32 = true;
//			for(int i=0; i<SIZE_BLOCK_AES; i++){
//				cipherTextBuf2[i] =
cipherText[i+(cipherText.length-SIZE_BLOCK_AES)];
//			}
//			cipherTextBuf = new byte[cipherText.length-
SIZE_BLOCK_AES];
//			for(int i=0; i<cipherTextBuf.length; i++){
//				cipherTextBuf[i] = cipherText[i];
//			}
//			cipherText = cipherTextBuf;
//		}

		int N = cipherText.length / (SIZE_BLOCK_AES*MAX_N_BLOCKS_AES);
//		if(ov32){
//			dataOut = new byte[cipherText.length+SIZE_BLOCK_AES];
//		}else{
			dataOut = new byte[cipherText.length];
//		}

		if(N>0){
			if(cipherText.length % (SIZE_BLOCK_AES*MAX_N_BLOCKS_AES)
!= 0){
				N++;
			}
			for(int i=0; i<N; i++){
				int a = i*SIZE_BLOCK_AES*MAX_N_BLOCKS_AES;
				int b = (i+1)*SIZE_BLOCK_AES*MAX_N_BLOCKS_AES;
				if(cipherText.length < b){
					cipherTextBuf = new byte[cipherText.length -
a];
					for(int j=0; j<(cipherText.length - a); j++){
						cipherTextBuf[j] = cipherText[j+a];
					}
				}else{
					cipherTextBuf = new byte[MAX_BYTES_AES];
					for(int j=0; j<MAX_BYTES_AES; j++){
						cipherTextBuf[j] =
cipherText[j+(i*MAX_BYTES_AES)];
					}
				}
				dataIn = concatenateArrayByte(key, cipherTextBuf);
				int P1 = (dataIn.length-
SIZE_KEY_AES)/SIZE_BLOCK_AES;
				try{
					dataOutBuf = sendAPDUwithData(CLA,
INS_AES_DECRYPT_V2, P1, 0, dataIn, channel);
					time += getTimeExecution();
				}catch(NullPointerException ex1){
					System.out.println(ex1.getMessage());
					System.exit(-1);
```

```
                            }
                            for(int j=0; j<dataOutBuf.length; j++){

        dataOut[j+(i*SIZE_BLOCK_AES*MAX_N_BLOCKS_AES)] = dataOutBuf[j];
                            }
                    }
            }else{
                    dataIn = concatenateArrayByte(key, cipherText);
                    int P1 = (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES;
                    // Revise Exception NullPointerException
                    dataOutBuf = sendAPDUwithData(CLA, INS_AES_DECRYPT_V2, P1,
0, dataIn, channel);
                    time = getTimeExecution();
                    for(int i=0; i<dataOutBuf.length; i++){
                            dataOut[i] = dataOutBuf[i];
                    }
            }
//          if(ov32){
//                  dataIn = concatenateArrayByte(key, cipherTextBuf2,
cipherTextBuf2.length);
//                  dataOutBuf = sendAPDUwithData(CLA, INS_AES_DECRYPT,
dataIn.length, 0, dataIn, channel);
//                  time += getTimeExecution();
//                  for(int i=0; i<SIZE_BLOCK_AES; i++){
//                          dataOut[i+(cipherText.length)] = dataOutBuf[i];
//                  }
//          }
            dataOut = deletePadding(dataOut);
            writeFile("C:/TMP/output-plainText", dataOut);
            disconnectCard(card);
            System.out.println("Decryption duration: "+time);
            System.out.println("Decryption succesful!");
    }

}/** End Decrypt2.java */
```

SHA1 class

```
package src2;
import java.io.UnsupportedEncodingException;

import javax.smartcardio.*;

public class SHA1 extends Functions{

        public static void main(String args[]){
                Card card = connectCard();
                CardChannel channel = establishChannel(card);

                byte[] dataOut;
                byte[] dataIn = {0x61, 0x62, 0x63};

                // dataIn has to be minor than 1024B !!
                int P1 = dataIn.length;
                System.out.println(P1+" Bytes sent");
                dataOut = sendAPDUwithData(CLA, INS_SHA1, 0, 0, dataIn,
channel);
                try{
                        String s = new String(dataOut, "UTF-8");
                        System.out.println(s);
                }catch(UnsupportedEncodingException ex1){
                        System.out.println(ex1.getMessage());
                        System.exit(-1);
                }
                System.out.println(getHexString(dataOut));

                disconnectCard(card);
        }
}
```

SHA256 class

```
package src2;
import javax.smartcardio.*;

public class SHA256 extends Functions{

        public static void main(String args[]){
                Card card = connectCard();
                CardChannel channel = establishChannel(card);

                byte[] dataIn, dataOut;
                dataIn = readFile("C:/TMP/input");
                // maximum size dataIn for SHA256?
                dataOut = sendAPDUwithData(CLA, INS_SHA256, dataIn.length, 0,
dataIn, channel);
                System.out.println(getHexString(dataOut));

                disconnectCard(card);
        }
}
```

SHA512 class

```
package src2;
import javax.smartcardio.*;

public class SHA512 extends Functions{

        public static void main(String args[]){
                Card card = connectCard();
                CardChannel channel = establishChannel(card);

                byte[] dataIn, dataOut;
                dataIn = readFile("C:/TMP/input");
                // maximum size dataIn for SHA512?
                dataOut = sendAPDUwithData(CLA, INS_SHA512, dataIn.length, 0,
dataIn, channel);
                System.out.println(getHexString(dataOut));

                disconnectCard(card);
        }
}
```

DetailsSC class

```
import java.io.UnsupportedEncodingException;

import javax.smartcardio.*;

public class DetailsSC extends Functions{

        public static void main(String args[]){
                Card card = connectCard();
                CardChannel channel = establishChannel(card);

                byte[] dataOut;
                dataOut = sendSimpleAPDUwithData(CLA, INS_DETAILS, channel);

                try{
                        String s = new String(dataOut, "UTF-8");
                        System.out.println(s);
                }catch(UnsupportedEncodingException ex1){
                        System.out.println(ex1.getMessage());
```

```
                    System.exit(-1);
                }

                disconnectCard(card);
        }

}
```

TestAPDU class

```
import javax.smartcardio.*;

public class TestAPDU extends Functions {

        public static void main(String args[]){
                Card card = connectCard();
                CardChannel channel = establishChannel(card);

                sendSimpleAPDU(CLA, INS_SIMPLE_APDU, channel);
                System.out.println("Time TestAPDU:
"+Functions.getTimeExecution());

                disconnectCard(card);
        }

}
```

NotEncrypt class

```
import java.io.UnsupportedEncodingException;
import javax.smartcardio.Card;
import javax.smartcardio.CardChannel;

public class NotEncrypt extends Functions{

    public static void main(String args[]){
       long time = 0;

       Card card = connectCard();
       CardChannel channel = establishChannel(card);

        byte[] key = readFile("C:/TMP/key");
        byte[] plainText = readFile("C:/TMP/input");

        byte[] dataIn, dataOut, plainTextBuf, dataOutBuf;
        dataOut = null;
        plainTextBuf = new byte[SIZE_BLOCK_AES];
        dataOutBuf = new byte[SIZE_BLOCK_AES];

        plainText = addPadding(plainText);
        dataOut = new byte[plainText.length];
        System.out.println("Length data to NOT encrypt: "+plainText.length);
        try{
            if(plainText.length>SIZE_BLOCK_AES){
             int N = plainText.length / SIZE_BLOCK_AES;
                for(int i=0; i<N; i++){
                    // take the segment of plainText to encrypt
                    for(int j=0; j<SIZE_BLOCK_AES; j++){
                        plainTextBuf[j] = plainText[j+(i*SIZE_BLOCK_AES)];
                    }
                    dataIn = concatenateArrayByte(key, plainTextBuf);
                    dataOutBuf = sendAPDUwithData(CLA, INS_AES_NOT_ENCRYPT,
dataIn.length, 0, dataIn, channel);
                    time += getTimeExecution();
                    for(int j=0; j<SIZE_BLOCK_AES; j++){
                        dataOut[j+(i*SIZE_BLOCK_AES)] = dataOutBuf[j];
                    }
                }
            }else{
                dataIn = concatenateArrayByte(key, plainText);
                dataOut = sendAPDUwithData(CLA, INS_AES_NOT_ENCRYPT,
dataIn.length, 0, dataIn, channel);
                time = getTimeExecution();
            }
        }catch(NullPointerException ex3){
            System.out.println(ex3.getMessage());
            System.exit(-1);
        }

        disconnectCard(card);
        System.out.println("NOT Encryption duration: "+time);
        System.out.println("NOT Encryption succesful!");
        try{
             System.out.println(new String(dataOut, "UTF-8"));
        }catch(UnsupportedEncodingException ex1){
                    System.out.println(ex1.getMessage());
             }
    }
}
/** End NotEncrypt.java */
```

NotEncrypt2 class

```
import javax.smartcardio.*;

public class NotEncrypt2 extends Functions{
        public static void main(String args[]){
                Card card = connectCard();
                CardChannel channel = establishChannel(card);
                long time = 0;

                byte[] plainText = readFile("C:/TMP/input");
                byte[] key = readFile("C:/TMP/key");
                byte[] dataIn, plainTextBuf, dataOut;

                plainText = addPadding(plainText);
                System.out.println("Length data sent (to encrypt):
"+plainText.length);

                int N = plainText.length / (SIZE_BLOCK_AES*MAX_N_BLOCKS_AES);
                if(N>0){
                        if(plainText.length % (SIZE_BLOCK_AES*MAX_N_BLOCKS_AES) !=
0){
                                N++;
                        }

                        for(int i=0; i<N; i++){
                                int a = i*SIZE_BLOCK_AES*MAX_N_BLOCKS_AES;
                                int b = (i+1)*SIZE_BLOCK_AES*MAX_N_BLOCKS_AES;
                                if(plainText.length < b){
                                        plainTextBuf = new byte[plainText.length -
a];
                                        for(int j=0; j<(plainText.length - a); j++){
                                                plainTextBuf[j] = plainText[j+a];
                                        }
                                }else{
                                        plainTextBuf = new byte[MAX_BYTES_AES];
                                        for(int j=0; j<MAX_BYTES_AES; j++){
                                                plainTextBuf[j] =
plainText[j+(i*MAX_BYTES_AES)];
                                        }
                                }
```

```
                          try{
                                dataIn = concatenateArrayByte(key, 
plainTextBuf);
                                int P1 = (dataIn.length-
SIZE_KEY_AES)/SIZE_BLOCK_AES;
                                System.out.println("P1: "+P1);
                                dataOut = sendAPDUwithData(CLA, 
INS_AES_NOT_ENCRYPT_V2, P1, 0, dataIn, channel);
                                if(plainTextBuf.length != dataOut.length){
                                      System.out.println("Error in the data 
field received, the length is not the same as expected");
                                      break;
                                }
                                time += getTimeExecution();
                          }catch(NullPointerException ex1){
                                System.out.println(ex1.getMessage());
                                System.exit(-1);
                          }
                    }
             }else{
                    dataIn = concatenateArrayByte(key, plainText);
                    int P1 = (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES;
                    System.out.println("P1: "+P1);
                    dataOut = sendAPDUwithData(CLA, INS_AES_NOT_ENCRYPT_V2, 
P1, 0, dataIn, channel);
                    if(plainText.length != dataOut.length){
                          System.out.println("Error in the data field 
received, the length is not the same as expected");
                    }
                    time += getTimeExecution();
             }
             
             System.out.println("Runtime APDU with data: "+time+" ms");
             disconnectCard(card);
      }
}
```

Functions class

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.List;
import javax.smartcardio.Card;
import javax.smartcardio.CardChannel;
import javax.smartcardio.CardException;
import javax.smartcardio.CardTerminal;
import javax.smartcardio.CommandAPDU;
import javax.smartcardio.ResponseAPDU;
import javax.smartcardio.TerminalFactory;

/**
 * Functions to use with the interface smart card
 * @author antonio
 *
 */
public class Functions {
      public static final int SIZE_BLOCK_AES = 16;
      public static final int SIZE_KEY_AES = 32;
      public static final int MAX_N_BLOCKS_AES = 6;
      public static final int MAX_BYTES_AES = 96;
      public static final int CLA = 128;
      public static final int INS_AES_ENCRYPT = 2;
      public static final int INS_AES_ENCRYPT_V2 = 8;
	public static final int INS_AES_DECRYPT = 6;
	public static final int INS_AES_DECRYPT_V2 = 7;
	public static final int INS_SHA1 = 2;
	public static final int INS_SHA256 = 2;
	public static final int INS_SHA512 = 2;
	public static final int INS_DETAILS = 12;
	public static final int INS_SIMPLE_APDU_WITHDATA = 5;
	public static final int INS_SIMPLE_APDU_WITHDATA_V2 = 9;
	public static final int INS_SIMPLE_APDU = 3;
	public static final int INS_AES_NOT_ENCRYPT = 13;
	private static long t1;
	private static long t2;

	/**
	 * Establish a connection with the smart card reader
	 * @return Card
	 */
	public static Card connectCard(){
		TerminalFactory factory;
		List<CardTerminal> terminals;
		CardTerminal terminal;
		Card card = null;
		try{
			factory = TerminalFactory.getDefault();
			terminals = factory.terminals().list();
			System.out.println("Terminals: "+terminals);
			terminal = terminals.get(0);
			card = terminal.connect("T=1");
			System.out.println("card :"+card);
		}catch(CardException ex1){
			System.out.println(ex1.getMessage());
			System.exit(-1);
		}
		return card;
	}


	/**
	 * Disconnect the card.
	 * @param card
	 */
	public static void disconnectCard(Card card){
		try{
			card.disconnect(false);
		}catch(CardException ex1){
			System.out.println(ex1.getMessage());
			System.exit(-1);
		}
	}


	/**
	 * Establish a channel with the smart card reader
	 * @return CardChannel
	 */
	public static CardChannel establishChannel(Card card){
		CardChannel channel = null;
		channel = card.getBasicChannel();
		return channel;
	}


	/**
	 * Add padding to a array of bytes until the array would
	 * be multiple of SIZE_BLOCK_AES.
	 * @param plainText
	 * @return byte[]
```

```
         */
        public static byte[] addPadding(byte[] plainText){
               int N = plainText.length / SIZE_BLOCK_AES;
          int resto = plainText.length % SIZE_BLOCK_AES;
          byte[] plainText2 = null;
          if(resto != 0){ // add padding
              int n2 = (N+1)*SIZE_BLOCK_AES;
              resto = n2 % plainText.length;
              plainText2 = new byte[n2];
              for(int i=0; i<plainText2.length; i++){
                  if(i<plainText.length){
                      plainText2[i] = plainText[i];
                  }else{
                      plainText2[i] = (byte) resto;
                  }
              }
          }else{
               plainText2 = plainText;
          }
          return plainText2;
        }


        /**
         * Deletes the padding in the dataOut array of bytes if there is
         * padding in the array. If there is not padding in the array,
         * returns the same dataOut array.
         * @param dataOut
         * @return byte[]
         */
        public static byte[] deletePadding(byte[] dataOut){
               boolean pad = false;
               int last = (int) dataOut[dataOut.length-1];
               if(last>0 && last<SIZE_BLOCK_AES){
                       pad = true;
                       for(int i=(dataOut.length-last); i<dataOut.length && pad;
i++){
                               if(dataOut[i] != last) pad = false;
                       }
               }
               if(!pad){
                       return dataOut;
               }else{
                 int N = dataOut.length - last;
                 byte[] dataOut2 = new byte[N];
                 for(int i=0; i<N; i++){
                     dataOut2[i] = dataOut[i];
                 }
                 return dataOut2;
               }
        }


        /**
         * Send a APDU command with data and returns the data of the
         * response APDU.
         * @param clas
         * @param ins
         * @param P1
         * @param P2
         * @param dataIn
         * @param channel
         * @return byte[]
         * @throws NullPointerException
         */
        public static byte[] sendAPDUwithData(int clas, int ins, int P1, int
P2, byte[] dataIn, CardChannel channel) throws NullPointerException{
```

```
              byte[] dataOut = null;
              ResponseAPDU r = null;
              CommandAPDU c = null;
              try{
                     c = new CommandAPDU(clas, ins, P1, P2, dataIn);
                     t1 = System.currentTimeMillis();
                     r = channel.transmit(c);
                     t2 = System.currentTimeMillis();
                     dataOut = r.getData();
              }catch(CardException ex1){
                     System.out.println("Error sendAPDUwithData
"+ex1.getMessage());
                     System.exit(-1);
              }
              if(r.getSW1() != 159){
                     NullPointerException ex2 = new NullPointerException("Error
with the APDU response, data returned invalid");
                     throw ex2;
              }else{
                     return dataOut;
              }
       }


       /**
        * Send a APDU command without data and returns the data of the
        * response APDU
        * @param clas
        * @param ins
        * @param channel
        * @return byte[]
        * @throws NullPointerException
        */
       public static byte[] sendSimpleAPDUwithData(int clas, int ins,
CardChannel channel) throws NullPointerException{
              byte[] dataOut = null;
              ResponseAPDU r = null;
              CommandAPDU c = null;
              try{
                     c = new CommandAPDU(clas, ins, 0, 0);
                     t1 = System.currentTimeMillis();
                     r = channel.transmit(c);
                     t2 = System.currentTimeMillis();
                     System.out.println(r);
                     dataOut = r.getData();
              }catch (CardException  ex1) {
                     System.out.println("Error sendAPDUwithData
"+ex1.getMessage());
                     System.exit(-1);
              }
              if(r.getSW1() != 159){
                     NullPointerException ex2 = new NullPointerException("Error
with the APDU response, data returned invalid");
                     throw ex2;
              }else{
                     return dataOut;
              }
       }

       /**
        * Returns the runtime of the command APDU sent to the smart card,
        * measured in the method sendAPDUwithData with the global
        * variables t1 and t2.
        * @return long
        */
       public static long getTimeExecution(){
              return t2-t1;
        }

        /**
         * Send a simple command APDU to the smart card
         * @param cla
         * @param ins
         * @param channel
         */
        public static void sendSimpleAPDU(int cla, int ins, CardChannel
channel){
                try{
                        CommandAPDU c = new CommandAPDU(cla, ins, 0, 0);
                        t1 = System.currentTimeMillis();
                        ResponseAPDU r = channel.transmit(c);
                        t2 = System.currentTimeMillis();
                        System.out.println(r);
                }catch(CardException ex1){
                        System.out.println(ex1.getMessage());
                        System.exit(-1);
                }
        }


        /**
         * Return an array of Bytes read from a fileName file
         * @param fileName
         * @return byte[]
         */
        public static byte[] readFile(String fileName){
         File file;
         FileInputStream fis;
         byte[] data_read = null;
         int read = -1;

         try{
             file = new File(fileName);
             fis = new FileInputStream(file);
             int N = fis.available();
             data_read = new byte[N];
             read = fis.read(data_read);
         }catch(FileNotFoundException ex1){
             System.out.println(ex1.getMessage());
         }catch(IOException ex2){
             System.out.println(ex2.getMessage());
         }catch(Exception ex3){
             System.out.println(ex3.getMessage());
         }
         if(read != -1){
             return data_read;
         }else{
             return null;
         }
     }


        /**
         * Writes an array of Bytes (data_write) to a fileName file
         * @param fileName
         * @param data_write
         */
        public static void writeFile(String fileName, byte[] data_write){
            File file;
            FileOutputStream fis;

            try{
                file = new File(fileName);
                fis = new FileOutputStream(file);
                fis.write(data_write);
            }catch(FileNotFoundException ex1){
                System.out.println(ex1.getMessage());
            }catch(IOException ex2){
                System.out.println(ex2.getMessage());
            }catch(Exception ex3){
                System.out.println(ex3.getMessage());
            }
        }

        /**
         * Returns an array of bytes with the key and the plain text
         * concatenated, ready to send in the data field of the command APDU.
         * @param key
         * @param plainText
         * @return byte[]
         */
    public static byte[] concatenateArrayByte(byte[] key, byte[] plainText){
        byte[] ret = new byte[SIZE_KEY_AES+plainText.length];
        int i;
        for(i=0; i<SIZE_KEY_AES; i++) ret[i] = key[i];
        for(i=SIZE_KEY_AES; i<(SIZE_KEY_AES+plainText.length); i++) ret[i] =
plainText[i-SIZE_KEY_AES];
        return ret;
    }


}/** End Functions.java */
```

Cryptographic applications on the smart card (AES encryption and decryption)

Source files

aes32.c

```
/*
*   Byte-oriented AES-256 implementation.
*   All lookup tables replaced with 'on the fly' calculations.
*
*   Copyright (c) 2007-2009 Ilya O. Levin, http://www.literatecode.com
*   Other contributors: Hal Finney
*
*   Permission to use, copy, modify, and distribute this software for any
*   purpose with or without fee is hereby granted, provided that the above
*   copyright notice and this permission notice appear in all copies.
*
*   THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
*   WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
*   MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
*   ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
*   WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
*   ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
*   OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/
#include "aes32.h"

#define F(x)    (((x)<<1) ^ ((((x)>>7) & 1) * 0x1b))
#define FD(x)   (((x) >> 1) ^ (((x) & 1) ? 0x8d : 0))

// #define BACK_TO_TABLES
#ifdef BACK_TO_TABLES

const uint8_t sbox[256] = {
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
    0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
    0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
    0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
    0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
    0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
    0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
    0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
    0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
    0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
    0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
    0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
    0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
    0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
    0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};
const uint8_t sboxinv[256] = {
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
    0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
    0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,
    0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,
    0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
    0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
    0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
    0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02,
    0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea,
    0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85,
    0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
    0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20,
    0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31,
    0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,
    0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,
    0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
    0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};
```

```
#define rj_sbox(x)     sbox[(x)]
#define rj_sbox_inv(x) sboxinv[(x)]

#else /* tableless subroutines */

/* --------------------------------------------------------------------------
*/
uint8_t gf_alog(uint8_t x) // calculate anti-logarithm gen 3
{
    uint8_t atb = 1, z;

    while (x--) {z = atb; atb <<= 1; if (z & 0x80) atb^= 0x1b; atb ^= z;}

    return atb;
} /* gf_alog */

/* --------------------------------------------------------------------------
*/
uint8_t gf_log(uint8_t x) // calculate logarithm gen 3
{
    uint8_t atb = 1, i = 0, z;

    do {
        if (atb == x) break;
        z = atb; atb <<= 1; if (z & 0x80) atb^= 0x1b; atb ^= z;
    } while (++i > 0);

    return i;
} /* gf_log */


/* --------------------------------------------------------------------------
*/
uint8_t gf_mulinv(uint8_t x) // calculate multiplicative inverse
{
    return (x) ? gf_alog(255 - gf_log(x)) : 0;
} /* gf_mulinv */

/* --------------------------------------------------------------------------
*/
uint8_t rj_sbox(uint8_t x)
{
    uint8_t y, sb;

    sb = y = gf_mulinv(x);
    y = (y<<1)|(y>>7); sb ^= y;  y = (y<<1)|(y>>7); sb ^= y;
    y = (y<<1)|(y>>7); sb ^= y;  y = (y<<1)|(y>>7); sb ^= y;

    return (sb ^ 0x63);
} /* rj_sbox */

/* --------------------------------------------------------------------------
*/
uint8_t rj_sbox_inv(uint8_t x)
{
    uint8_t y, sb;

    y = x ^ 0x63;
    sb = y = (y<<1)|(y>>7);
    y = (y<<2)|(y>>6); sb ^= y; y = (y<<3)|(y>>5); sb ^= y;

    return gf_mulinv(sb);
} /* rj_sbox_inv */

#endif
```

```
/* ---------------------------------------------------------------------------
*/
uint8_t rj_xtime(uint8_t x)
{
    return (x & 0x80) ? ((x << 1) ^ 0x1b) : (x << 1);
} /* rj_xtime */

/* ---------------------------------------------------------------------------
*/
void aes_subBytes(uint8_t *buf)
{
    register uint8_t i = 16;

    while (i--) buf[i] = rj_sbox(buf[i]);
} /* aes_subBytes */

/* ---------------------------------------------------------------------------
*/
void aes_subBytes_inv(uint8_t *buf)
{
    register uint8_t i = 16;

    while (i--) buf[i] = rj_sbox_inv(buf[i]);
} /* aes_subBytes_inv */

/* ---------------------------------------------------------------------------
*/
void aes_addRoundKey(uint8_t *buf, uint8_t *key)
{
    register uint8_t i = 16;

    while (i--) buf[i] ^= key[i];
} /* aes_addRoundKey */

/* ---------------------------------------------------------------------------
*/
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)  buf[i] ^= (cpk[i] = key[i]), cpk[16+i] = key[16 + i];
} /* aes_addRoundKey_cpy */


/* ---------------------------------------------------------------------------
*/
void aes_shiftRows(uint8_t *buf)
{
    register uint8_t i, j; /* to make it potentially parallelable :) */

    i = buf[1]; buf[1] = buf[5]; buf[5] = buf[9]; buf[9] = buf[13]; buf[13] =
i;
    i = buf[10]; buf[10] = buf[2]; buf[2] = i;
    j = buf[3]; buf[3] = buf[15]; buf[15] = buf[11]; buf[11] = buf[7]; buf[7]
= j;
    j = buf[14]; buf[14] = buf[6]; buf[6]  = j;

} /* aes_shiftRows */

/* ---------------------------------------------------------------------------
*/
void aes_shiftRows_inv(uint8_t *buf)
{
    register uint8_t i, j; /* same as above :) */

    i = buf[1]; buf[1] = buf[13]; buf[13] = buf[9]; buf[9] = buf[5]; buf[5] =
i;
```

```
    i = buf[2]; buf[2] = buf[10]; buf[10] = i;
    j = buf[3]; buf[3] = buf[7]; buf[7] = buf[11]; buf[11] = buf[15]; buf[15]
= j;
    j = buf[6]; buf[6] = buf[14]; buf[14] = j;

} /* aes_shiftRows_inv */

/* ---------------------------------------------------------------------------
*/
void aes_mixColumns(uint8_t *buf)
{
    register uint8_t i, a, b, c, d, e;

    for (i = 0; i < 16; i += 4)
    {
        a = buf[i]; b = buf[i + 1]; c = buf[i + 2]; d = buf[i + 3];
        e = a ^ b ^ c ^ d;
        buf[i] ^= e ^ rj_xtime(a^b);   buf[i+1] ^= e ^ rj_xtime(b^c);
        buf[i+2] ^= e ^ rj_xtime(c^d); buf[i+3] ^= e ^ rj_xtime(d^a);
    }
} /* aes_mixColumns */

/* ---------------------------------------------------------------------------
*/
void aes_mixColumns_inv(uint8_t *buf)
{
    register uint8_t i, a, b, c, d, e, x, y, z;

    for (i = 0; i < 16; i += 4)
    {
        a = buf[i]; b = buf[i + 1]; c = buf[i + 2]; d = buf[i + 3];
        e = a ^ b ^ c ^ d;
        z = rj_xtime(e);
        x = e ^ rj_xtime(rj_xtime(z^a^c));  y = e ^ rj_xtime(rj_xtime(z^b^d));
        buf[i] ^= x ^ rj_xtime(a^b);   buf[i+1] ^= y ^ rj_xtime(b^c);
        buf[i+2] ^= x ^ rj_xtime(c^d); buf[i+3] ^= y ^ rj_xtime(d^a);
    }
} /* aes_mixColumns_inv */

/* ---------------------------------------------------------------------------
*/
void aes_expandEncKey(uint8_t *k, uint8_t *rc)
{
    register uint8_t i;

    k[0] ^= rj_sbox(k[29]) ^ (*rc);
    k[1] ^= rj_sbox(k[30]);
    k[2] ^= rj_sbox(k[31]);
    k[3] ^= rj_sbox(k[28]);
    *rc = F( *rc);

    for(i = 4; i < 16; i += 4)  k[i] ^= k[i-4],   k[i+1] ^= k[i-3],
        k[i+2] ^= k[i-2], k[i+3] ^= k[i-1];
    k[16] ^= rj_sbox(k[12]);
    k[17] ^= rj_sbox(k[13]);
    k[18] ^= rj_sbox(k[14]);
    k[19] ^= rj_sbox(k[15]);

    for(i = 20; i < 32; i += 4) k[i] ^= k[i-4],   k[i+1] ^= k[i-3],
        k[i+2] ^= k[i-2], k[i+3] ^= k[i-1];

} /* aes_expandEncKey */

/* ---------------------------------------------------------------------------
*/
void aes_expandDecKey(uint8_t *k, uint8_t *rc)
{
    uint8_t i;

    for(i = 28; i > 16; i -= 4) k[i+0] ^= k[i-4], k[i+1] ^= k[i-3],
        k[i+2] ^= k[i-2], k[i+3] ^= k[i-1];

    k[16] ^= rj_sbox(k[12]);
    k[17] ^= rj_sbox(k[13]);
    k[18] ^= rj_sbox(k[14]);
    k[19] ^= rj_sbox(k[15]);

    for(i = 12; i > 0; i -= 4)  k[i+0] ^= k[i-4], k[i+1] ^= k[i-3],
        k[i+2] ^= k[i-2], k[i+3] ^= k[i-1];

    *rc = FD(*rc);
    k[0] ^= rj_sbox(k[29]) ^ (*rc);
    k[1] ^= rj_sbox(k[30]);
    k[2] ^= rj_sbox(k[31]);
    k[3] ^= rj_sbox(k[28]);
} /* aes_expandDecKey */


/* -------------------------------------------------------------------------
*/
void aes256_init(aes256_context *ctx, uint8_t *k)
{
    uint8_t rcon = 1;
    register uint8_t i;

    for (i = 0; i < sizeof(ctx->key); i++) ctx->enckey[i] = ctx->deckey[i] =
k[i];
    for (i = 8;--i;) aes_expandEncKey(ctx->deckey, &rcon);
} /* aes256_init */

/* -------------------------------------------------------------------------
*/
void aes256_done(aes256_context *ctx)
{
    register uint8_t i;

    for (i = 0; i < sizeof(ctx->key); i++)
        ctx->key[i] = ctx->enckey[i] = ctx->deckey[i] = 0;
} /* aes256_done */

/* -------------------------------------------------------------------------
*/
void aes256_encrypt_ecb(aes256_context *ctx, uint8_t *buf)
{
    uint8_t i, rcon;

    aes_addRoundKey_cpy(buf, ctx->enckey, ctx->key);
    for(i = 1, rcon = 1; i < 14; ++i)
    {
        aes_subBytes(buf);
        aes_shiftRows(buf);
        aes_mixColumns(buf);
        if( i & 1 ) aes_addRoundKey( buf, &ctx->key[16]);
        else aes_expandEncKey(ctx->key, &rcon), aes_addRoundKey(buf, ctx-
>key);
    }
    aes_subBytes(buf);
    aes_shiftRows(buf);
    aes_expandEncKey(ctx->key, &rcon);
    aes_addRoundKey(buf, ctx->key);
} /* aes256_encrypt */

/* -------------------------------------------------------------------------
*/
```

```
void aes256_decrypt_ecb(aes256_context *ctx, uint8_t *buf)
{
    uint8_t i, rcon;

    aes_addRoundKey_cpy(buf, ctx->deckey, ctx->key);
    aes_shiftRows_inv(buf);
    aes_subBytes_inv(buf);

    for (i = 14, rcon = 0x80; --i;)
    {
        if( ( i & 1 ) )
        {
            aes_expandDecKey(ctx->key, &rcon);
            aes_addRoundKey(buf, &ctx->key[16]);
        }
        else aes_addRoundKey(buf, ctx->key);
        aes_mixColumns_inv(buf);
        aes_shiftRows_inv(buf);
        aes_subBytes_inv(buf);
    }
    aes_addRoundKey( buf, ctx->key);
} /* aes256_decrypt */
```

functions_smartcard.c

```
/********************************************************
*                                                        *
*   This is the implementation of a basic smart card *
*   OS supporting the T=1 protokoll.                         *
*                                                        *
*********************************************************
*****************************************
* File: functions_smartcard.c                  *
*****************************************/

#include <string.h>
#include "functions_smartcard.h"
#include "globals.h"
#include "T1_Comm_Lib.h"
#include "aes32.h"

//#include <avr\sfr_defs.h>
#include <avr\io.h>
#include <avr\iom163.h>
#include <compat/deprecated.h>



//*****************************************************************
// Routine performes AES encryption on an SIZE_BLOCK_AES byte input block.
// Can be used for AES encrypt and internal authenticate
// CLA=0x80         INS=0x02
// P1=length key+plainText(dataIn)
void do_AES_Encrypt(command_APDU *com_APDU, response_APDU *resp_APDU){
      // Extract dataIn from *com_APDU
      // Call function encrypt with the dataIn and dataOut
    aes256_context ctx;
    uint8_t key[SIZE_KEY_AES];
    uint8_t buf[SIZE_BLOCK_AES];
      uint8_t i;

      for(i=0; i<SIZE_KEY_AES; i++){
            key[i] = (*com_APDU).data_field[i];
      }
      for(i=0; i<SIZE_BLOCK_AES; i++){
            buf[i] = (*com_APDU).data_field[i+SIZE_KEY_AES];
      }
```

```
       aes256_init(&ctx, key);
    aes256_encrypt_ecb(&ctx, buf);
       aes256_done(&ctx);

       for(i=0; i<SIZE_BLOCK_AES; i++){
              (*resp_APDU).data_field[i] = buf[i];
       }
       (*resp_APDU).LE = SIZE_BLOCK_AES;
       (*resp_APDU).LEN = SIZE_BLOCK_AES+SIZE_SW_RESPONSE;
       (*resp_APDU).SW1 = SW1_SUCCESS; // 0x9F
       (*resp_APDU).SW2 = SIZE_BLOCK_AES;
}
//*****************************************************************


//*****************************************************************
// Routine to check some parts of command APDU
// CLA=0x80         INS=0x03
void test_command(command_APDU *com_APDU, response_APDU *resp_APDU){
       (*resp_APDU).LEN = SIZE_SW_RESPONSE;
       (*resp_APDU).LE = 0;
       (*resp_APDU).SW1 = 0x90;
       (*resp_APDU).SW2 = 0x00;
}
//*****************************************************************


//*****************************************************************
// Routine to check the execution of the encryption code
// CLA=0x80         INS=0x04
// P1=size_dataIn
void test_timer0(command_APDU *com_APDU, response_APDU *resp_APDU){
       uint8_t N, i;
       uint8_t cal_ref_timer;
       uint8_t cal_ref_timer2;
       uint8_t overflow;
       uint8_t buffer[(*com_APDU).P1];

       N = (*com_APDU).P1;
       if(N>0 && N<15){
              (*resp_APDU).LEN = 2;
              (*resp_APDU).LE = 0;
              (*resp_APDU).SW1 = 0x6A;
              (*resp_APDU).SW2 = 0x88;
       }else{
              overflow = 0;
              DDRB = 0xFF;
              TCNT0 = 0;
              TCCR0 = 1;
              cal_ref_timer = TCNT0;
              for(i=0; i<N; i++){
                     buffer[i] = (*com_APDU).data_field[i];
              }
              // check if the the overflow bit of the TIFR register
              // is set.
              cal_ref_timer2 = TCNT0;
              if((TIFR&0x01) == 0x01){
                     overflow = 1;
                     cal_ref_timer2 = 0x00;
              }

              (*resp_APDU).LEN = 3+SIZE_SW_RESPONSE;
              (*resp_APDU).LE = 3;
              (*resp_APDU).SW1 = SW1_SUCCESS;
              (*resp_APDU).SW2 = 3;
              (*resp_APDU).data_field[0] = cal_ref_timer;
              (*resp_APDU).data_field[1] = cal_ref_timer2;
              (*resp_APDU).data_field[2] = overflow;
       }
}
//******************************************************************


//******************************************************************
// Routine to check some parts of command APDU
// CLA=0x80         INS=0x05
// P1=size_dataIn
void test_command_withData(command_APDU *com_APDU, response_APDU *resp_APDU){
       int N = (*com_APDU).P1;

       for(int i=0; i<N; i++){
              (*resp_APDU).data_field[i] = (*com_APDU).data_field[i];
       }

       (*resp_APDU).LE = N;
       (*resp_APDU).LEN = N+SIZE_SW_RESPONSE;
       (*resp_APDU).SW1 = SW1_SUCCESS;
       (*resp_APDU).SW2 = N;
}
//******************************************************************


//******************************************************************
// Routine performes AES decryption on an SIZE_BLOCK_AES byte input block.
// CLA=0x80         INS=0x06
// P1=length key+plainText(dataIn)
void do_AES_Decrypt(command_APDU *com_APDU, response_APDU *resp_APDU){
       // Extract dataIn from *com_APDU
       // Call function encrypt with the dataIn and dataOut
    aes256_context ctx;
    uint8_t key[SIZE_KEY_AES];
    uint8_t buf[SIZE_BLOCK_AES], i;

       for(i=0; i<SIZE_KEY_AES; i++){
              key[i] = (*com_APDU).data_field[i];
       }
       for(i=0; i<SIZE_BLOCK_AES; i++){
              buf[i] = (*com_APDU).data_field[i+SIZE_KEY_AES];
       }

       aes256_init(&ctx, key);
    aes256_decrypt_ecb(&ctx, buf);
       aes256_done(&ctx);

       for(i=0; i<SIZE_BLOCK_AES; i++){
              (*resp_APDU).data_field[i] = buf[i];
       }
       (*resp_APDU).LE = SIZE_BLOCK_AES;
       (*resp_APDU).LEN = SIZE_BLOCK_AES+SIZE_SW_RESPONSE;
       (*resp_APDU).SW1 = SW1_SUCCESS;
       (*resp_APDU).SW2 = SIZE_BLOCK_AES;

}
//******************************************************************


//******************************************************************
// Routine performes AES decryption on an SIZE_BLOCK_AES byte input block.
// CLA=0x80         INS=0x07
// P1= (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES
void do_AES_Decrypt2(command_APDU *com_APDU, response_APDU *resp_APDU){
    aes256_context ctx;
    uint8_t key[SIZE_KEY_AES];
    uint8_t buf[SIZE_BLOCK_AES], i, j, N;

       N = (*com_APDU).P1;
       for(i=0; i<SIZE_KEY_AES; i++){
              key[i] = (*com_APDU).data_field[i];
       }
       if(N > MAX_N_BLOCKS_AES){
              // The parameters in the data field are incorrect
              (*resp_APDU).LE = 0;
              (*resp_APDU).LEN = SIZE_SW_RESPONSE;
              (*resp_APDU).SW1 = 0x6A;
              (*resp_APDU).SW2 = 0x80;
       }else{
              aes256_init(&ctx, key);
              for(i=0; i<N; i++){
                     for(j=0; j<SIZE_BLOCK_AES; j++){
                            buf[j] =
(*com_APDU).data_field[SIZE_KEY_AES+(i*SIZE_BLOCK_AES)+j];
                     }
                  aes256_decrypt_ecb(&ctx, buf); // Decrypt data
                     for(j=0; j<SIZE_BLOCK_AES; j++){
                            (*resp_APDU).data_field[(i*SIZE_BLOCK_AES)+j] =
buf[j];
                     }
              }
              aes256_done(&ctx);

              (*resp_APDU).LE = SIZE_BLOCK_AES*N;
              (*resp_APDU).LEN = (SIZE_BLOCK_AES*N)+SIZE_SW_RESPONSE;
              (*resp_APDU).SW1 = SW1_SUCCESS;
              (*resp_APDU).SW2 = SIZE_BLOCK_AES*N;
       }
}
//*******************************************************************


//*******************************************************************
// Routine performes AES encryption on an SIZE_BLOCK_AES byte input block
// More data transmit in a APDU, improved
// Can be used for AES encrypt and internal authenticate
// CLA=0x80         INS=0x08
// P1 = (dataIn.length-SIZE_KEY_AES)/SIZE_BLOCK_AES;
void do_AES_Encrypt2(command_APDU *com_APDU, response_APDU *resp_APDU){
    aes256_context ctx;
    uint8_t key[SIZE_KEY_AES];
    uint8_t buf[SIZE_BLOCK_AES], i, j, N;

       N = (*com_APDU).P1;
       if(N > MAX_N_BLOCKS_AES){
              // The parameters in the data field are incorrect
              (*resp_APDU).LE = 0;
              (*resp_APDU).LEN = SIZE_SW_RESPONSE;
              (*resp_APDU).SW1 = 0x6A;
              (*resp_APDU).SW2 = 0x80;
       }else{
              for(i=0; i<SIZE_KEY_AES; i++){
                     key[i] = (*com_APDU).data_field[i];
              }
              aes256_init(&ctx, key);
              for(i=0; i<N; i++){
                     for(j=0; j<SIZE_BLOCK_AES; j++){
                            buf[j] =
(*com_APDU).data_field[SIZE_KEY_AES+(i*SIZE_BLOCK_AES)+j];
                     }
              aes256_encrypt_ecb(&ctx, buf); // Encrypt data
                     for(j=0; j<SIZE_BLOCK_AES; j++){
                            (*resp_APDU).data_field[(i*SIZE_BLOCK_AES)+j] =
buf[j];
                        }
                }
                aes256_done(&ctx);

                (*resp_APDU).LE = SIZE_BLOCK_AES*N;
                (*resp_APDU).LEN = (SIZE_BLOCK_AES*N)+SIZE_SW_RESPONSE;
                (*resp_APDU).SW1 = SW1_SUCCESS;
                (*resp_APDU).SW2 = SIZE_BLOCK_AES*N;
        }
}
//*****************************************************************


//*****************************************************************
// Routine to check some parts of command APDU
// CLA=0x80         INS=0x09
// P1= number blocks to copy (like Encrypt2)
void do_AES_NotEncrypt2(command_APDU *com_APDU, response_APDU *resp_APDU){
    uint8_t key[SIZE_KEY_AES];
    uint8_t buf[SIZE_BLOCK_AES], i, j, N;

        N = (*com_APDU).P1;
        if(N > MAX_N_BLOCKS_AES){
                // The parameters in the data field are incorrect
                (*resp_APDU).LE = 0;
                (*resp_APDU).LEN = SIZE_SW_RESPONSE;
                (*resp_APDU).SW1 = 0x6A;
                (*resp_APDU).SW2 = 0x80;
        }else{
                for(i=0; i<SIZE_KEY_AES; i++){
                        key[i] = (*com_APDU).data_field[i];
                }
                for(i=0; i<N; i++){
                        for(j=0; j<SIZE_BLOCK_AES; j++){
                                buf[j] =
(*com_APDU).data_field[SIZE_KEY_AES+(i*SIZE_BLOCK_AES)+j];
                        }

                        for(j=0; j<SIZE_BLOCK_AES; j++){
                                (*resp_APDU).data_field[(i*SIZE_BLOCK_AES)+j] =
buf[j];
                        }
                }
                (*resp_APDU).LE = SIZE_BLOCK_AES*N;
                (*resp_APDU).LEN = (SIZE_BLOCK_AES*N)+SIZE_SW_RESPONSE;
                (*resp_APDU).SW1 = SW1_SUCCESS;
                (*resp_APDU).SW2 = SIZE_BLOCK_AES*N;
        }
}
//*****************************************************************


//*****************************************************************
// Routine to check the execution of the encryption code
// CLA=0x80         INS=0x0A
// P1=size_dataIn
void test_timer1(command_APDU *com_APDU, response_APDU *resp_APDU){
        uint8_t N, i, overflow;
        uint8_t cal_ref_timerL, cal_ref_timerH;
        uint8_t cal_ref_timer2L, cal_ref_timer2H;
        uint8_t buffer[(*com_APDU).P1];

        N = (*com_APDU).P1;
        if(N>0 && N<15){
                (*resp_APDU).LEN = 2;
                (*resp_APDU).LE = 0;
                (*resp_APDU).SW1 = 0x6A;
                (*resp_APDU).SW2 = 0x88;
        }else{
                overflow = 0;
                DDRB = 0xFF; //     use all pins on PORTB for output
                TCNT1L = 0x00;      //     start value of T/C1 - low byte
                TCNT1H = 0x00;      //     start value of T/C1 - high byte
                TCCR1A = 0;         //     T/C1 in timer mode
                TCCR1B = 1;         //     prescale ck

                cal_ref_timerL = TCNT1L;
                cal_ref_timerH = TCNT1H;

                // Code to measure
                for(i=0; i<N; i++){
                        buffer[i] = (*com_APDU).data_field[i];
                }

                cal_ref_timer2L = TCNT1L;
                cal_ref_timer2H = TCNT1H;

                if((TIFR&0x04) == 0x04){
                        overflow = 1;
                        cal_ref_timer2L = 0x00;
                        cal_ref_timer2H = 0x00;
                }

                (*resp_APDU).LEN = 7;      // 2+5
                (*resp_APDU).LE = 5;
                (*resp_APDU).SW1 = SW1_SUCCESS; // 0x9F
                (*resp_APDU).SW2 = 5;
                (*resp_APDU).data_field[0] = cal_ref_timerH;
                (*resp_APDU).data_field[1] = cal_ref_timerL;
                (*resp_APDU).data_field[2] = overflow;
                (*resp_APDU).data_field[3] = cal_ref_timer2H;
                (*resp_APDU).data_field[4] = cal_ref_timer2L;
        }
}
//******************************************************************


//******************************************************************
// Routine performes AES encryption on an SIZE_BLOCK_AES byte input block.
// Can be used for AES encrypt and internal authenticate
// CLA=0x80         INS=0x0B
// P1=length key+plainText(dataIn)
void do_AES_Encrypt_withTimers(command_APDU *com_APDU, response_APDU
*resp_APDU){
        // Extract dataIn from *com_APDU
        // Call function encrypt with the dataIn and dataOut
    aes256_context ctx;
    uint8_t key[SIZE_KEY_AES];
    uint8_t buf[SIZE_BLOCK_AES];
        uint8_t i;

        for(i=0; i<SIZE_KEY_AES; i++){
                key[i] = (*com_APDU).data_field[i];
        }
        for(i=0; i<SIZE_BLOCK_AES; i++){
                buf[i] = (*com_APDU).data_field[i+SIZE_KEY_AES];
        }
        DDRB = 0xFF; //     use all pins on PORTB for output
        TCNT1L = 0x00;      //     start value of T/C1 - low byte
        TCNT1H = 0x00;      //     start value of T/C1 - high byte
        TCCR1A = 0;         //     T/C1 in timer mode
        TCCR1B = 5;         //     prescale ck
        (*resp_APDU).data_field[18] = 0;
        (*resp_APDU).data_field[17] = TCNT1L;
        (*resp_APDU).data_field[SIZE_BLOCK_AES] = TCNT1H;

        // Code to measure
        aes256_init(&ctx, key);
    aes256_encrypt_ecb(&ctx, buf);
        aes256_done(&ctx);


        if((TIFR&0x04) != 0x04){
                (*resp_APDU).data_field[20] = TCNT1L;   //     cal_ref_timer2L
                (*resp_APDU).data_field[19] = TCNT1H;   //     cal_ref_timer2H
        }else{
                (*resp_APDU).data_field[18] = 1;        //     overflow
                (*resp_APDU).data_field[20] = 0x00;            //
        cal_ref_timer2L
                (*resp_APDU).data_field[19] = 0x00;            //
        cal_ref_timer2H
        }

        for(i=0; i<SIZE_BLOCK_AES; i++){
                (*resp_APDU).data_field[i] = buf[i];
        }
        (*resp_APDU).LE = 21;
        (*resp_APDU).LEN = 21+SIZE_SW_RESPONSE;
        (*resp_APDU).SW1 = SW1_SUCCESS;
        (*resp_APDU).SW2 = 21;
}
//*******************************************************************


//*******************************************************************
// Return a string with the main application of the smart card
// CLA=0x80         INS=0x0C
void detailsApps(command_APDU *com_APDU, response_APDU *resp_APDU){
        uint8_t name[29] = {0x41, 0x45, 0x53, 0x20, 0x45, 0x6e, 0x63, 0x72,
0x79, 0x70, 0x74, 0x69, 0x6f, 0x6e, 0x20, 0x61, 0x6e, 0x64, 0x20, 0x44, 0x65,
0x63, 0x72, 0x79, 0x70, 0x74, 0x69, 0x6f, 0x6e};
        uint8_t i;

        for(i=0; i<29; i++){
                (*resp_APDU).data_field[i] = name[i];
        }
        (*resp_APDU).LE = 29;
        (*resp_APDU).LEN = 31;
        (*resp_APDU).SW1 = SW1_SUCCESS;
        (*resp_APDU).SW2 = 29;
}


//*******************************************************************
// Routine to check some parts of command APDU
// CLA=0x80         INS=0x0D
// P1=size_dataIn
void do_AES_NotEncrypt(command_APDU *com_APDU, response_APDU *resp_APDU){
    uint8_t key[SIZE_KEY_AES];
    uint8_t buf[SIZE_BLOCK_AES];
        uint8_t i;

        for(i=0; i<SIZE_KEY_AES; i++){
                key[i] = (*com_APDU).data_field[i];
        }
        for(i=0; i<SIZE_BLOCK_AES; i++){
                buf[i] = (*com_APDU).data_field[i+SIZE_KEY_AES];
        }
        for(i=0; i<SIZE_BLOCK_AES; i++){
                (*resp_APDU).data_field[i] = buf[i];
        }

        (*resp_APDU).LE = SIZE_BLOCK_AES;
        (*resp_APDU).LEN = SIZE_BLOCK_AES+SIZE_SW_RESPONSE;
        (*resp_APDU).SW1 = SW1_SUCCESS; // 0x9F
        (*resp_APDU).SW2 = SIZE_BLOCK_AES;
}
//*******************************************************************





//*******************************************************************
// Main command Handler
void command_Handler(command_APDU *com_APDU, response_APDU *resp_APDU){
    (*resp_APDU).NAD = (*com_APDU).NAD;
    (*resp_APDU).PCB = (*com_APDU).PCB;

    if((*com_APDU).PCB == 0xC1)              // S-Block Handling
      {
        (*resp_APDU).NAD = (*com_APDU).NAD;
        (*resp_APDU).PCB = 0xE1;
        (*resp_APDU).LEN = 1;
        (*resp_APDU).data_field[0] = (*com_APDU).CLA;
      }
    else                                    // I-Block Handling
      {
        switch((*com_APDU).CLA){
            case 0x80:{
                switch((*com_APDU).INS){
                    case 0x02:
                                   // Encrypt function
                          do_AES_Encrypt(com_APDU, resp_APDU);
                          break;
                                 case 0x03:
                                        // test command function
                                        test_command(com_APDU, resp_APDU);
                                        break;
                                 case 0x04:
                                        // function test timer/counter0
                                        test_timer0(com_APDU, resp_APDU);
                                        break;
                                 case 0x05:
                                        // test command function with data
                                        test_command_withData(com_APDU,
resp_APDU);
                                        break;
                                 case 0x06:
                                        // Decrypt function
                                        do_AES_Decrypt(com_APDU, resp_APDU);
                                        break;
                                 case 0x07:
                                        // Execute Decrypt2 AES function
(improved)
                                        do_AES_Decrypt2(com_APDU, resp_APDU);
                                        break;
                                 case 0x08:
                                        // Execute Encrypt2 AES function
(improved)
                                        do_AES_Encrypt2(com_APDU, resp_APDU);
                                        break;
                                 case 0x09:
                                        // Execute Encrypt2 AES function
without Encrypt code
                                        do_AES_NotEncrypt2(com_APDU,
resp_APDU);
                                        break;
                                case 0x0A:
                                        // function test timer/counter1
                                        test_timer1(com_APDU, resp_APDU);
                                        break;
                                case 0x0B:
                                        // Encrypt function with timers
                                        do_AES_Encrypt_withTimers(com_APDU,
resp_APDU);
                                        break;
                                case 0x0C:
                                        // Return a string with the main
application of the smart card
                                        detailsApps(com_APDU, resp_APDU);
                                        break;
                                case 0x0D:
                                        // Execute Encrypt AES function
without Encrypt code
                                        do_AES_NotEncrypt(com_APDU,
resp_APDU);
                                        break;
                    default:
                        (*resp_APDU).LEN = 2;
                        (*resp_APDU).LE = 0;
                        (*resp_APDU).SW1 = 0x6d;
                       (*resp_APDU).SW2 = 0x00;
                                        // Command not allowed. Invalid
instruction byte (INS)
                        break;
                  }
                break;
              }
            default:
              {
                (*resp_APDU).LEN = 2;
                (*resp_APDU).LE = 0;
                (*resp_APDU).SW1 = 0x6e;
                (*resp_APDU).SW2 = 0x00;
                        // Incorrect application (CLA parameter of a
command)
                break;
              }
          }
      }
  }
//*******************************************************************
```

main.c

```
/*********************************************************
*                                                       *
*   This is the implementation of a basic smart card *
*   OS supporting the T=1 protokoll.                             *
*                                                       *
*********************************************************
*****************************************
* File: main.c                                             *
*****************************************/

#include "globals.h"
#include "T1_Comm_Lib.h"
#include "functions_smartcard.h"


void main(void)
{
	unsigned char result, i;
	command_APDU rec_APDU;		// struct for command APDU
	response_APDU res_APDU;	// struct for response APDU

	for(i=0;i<50;i++);		// wait with the ATR at least 400 cycles
		send_ATR();			// send the Answer to Reset according 
to ISO

	// endless loop to receive/respond and process commands
	for(;;)
	{
		result = receive_APDU(&rec_APDU);	// receive APDU according to 
T=1

		if(result != OK)		// if EDC Checksum error return 
error code
		{
			res_APDU.NAD = rec_APDU.NAD;
			res_APDU.PCB = rec_APDU.PCB;
			res_APDU.LEN = 2;
			res_APDU.LE = 0;
			res_APDU.SW1 = 0x67;
			res_APDU.SW2 = 0x01;
		}else{
			command_Handler(&rec_APDU, &res_APDU);
		}

		send_APDU(&res_APDU);	// send the result APDU

		// Reset all APDUS
		res_APDU.NAD = 0x00;
		res_APDU.PCB = 0x00;
		res_APDU.LEN = 2;
		res_APDU.LE = 0;
		res_APDU.SW1 = 0x64;
		res_APDU.SW2 = 0x00;

		rec_APDU.NAD = 0x00;
		rec_APDU.PCB = 0x00;
		rec_APDU.LEN = 0;
		rec_APDU.LE = 0;
		rec_APDU.LC = 0;
		rec_APDU.CLA = 0x00;
		rec_APDU.INS = 0x00;
	}
}
```

Header files

aes32.h

```
/*
*   Byte-oriented AES-256 implementation.
*   All lookup tables replaced with 'on the fly' calculations.
*
*   Copyright (c) 2007-2009 Ilya O. Levin, http://www.literatecode.com
*   Other contributors: Hal Finney
*
*   Permission to use, copy, modify, and distribute this software for any
*   purpose with or without fee is hereby granted, provided that the above
*   copyright notice and this permission notice appear in all copies.
*
*   THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
*   WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
*   MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
*   ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
*   WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
*   ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
*   OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/

#include <inttypes.h>


typedef struct {
        uint8_t key[32];
        uint8_t enckey[32];
        uint8_t deckey[32];
} aes256_context;


void aes256_init(aes256_context *, uint8_t * /* key */);
void aes256_done(aes256_context *);
void aes256_encrypt_ecb(aes256_context *, uint8_t * /* plaintext */);
void aes256_decrypt_ecb(aes256_context *, uint8_t * /* cipertext */);
```

functions_smartcard.h

```
/*********************************************************
* IAIK * IAIK * IAIK * IAIK * IAIK * IAIK * IAIK * IAIK *
*                                                       *
*   This is the implementation of a basic smart card *
*   OS supporting the T=1 protokoll.                          *
*                                                       *
*********************************************************
*****************************************
* File: functions_smartcard.h                *
*****************************************/

#ifndef _IAIK_SC_OS_functions
#define _IAIK_SC_OS_functions

#include "globals.h"

// update !!

void do_AES_Encrypt(command_APDU *com_APDU, response_APDU *resp_APDU);
void test_command(command_APDU *com_APDU, response_APDU *resp_APDU);
void test_timer0(command_APDU *com_APDU, response_APDU *resp_APDU);
void test_command_withData(command_APDU *com_APDU, response_APDU *resp_APDU);
void do_AES_Decrypt(command_APDU *com_APDU, response_APDU *resp_APDU);
void do_AES_Decrypt2(command_APDU *com_APDU, response_APDU *resp_APDU);
void do_AES_Encrypt2(command_APDU *com_APDU, response_APDU *resp_APDU);
void do_AES_NotEncrypt2(command_APDU *com_APDU, response_APDU *resp_APDU);
void test_timer1(command_APDU *com_APDU, response_APDU *resp_APDU);
void do_AES_Encrypt_withTimers(command_APDU *com_APDU, response_APDU
*resp_APDU);
void detailsApps(command_APDU *com_APDU, response_APDU *resp_APDU);
void do_AES_NotEncrypt(command_APDU *com_APDU, response_APDU *resp_APDU);
void command_Handler(command_APDU *com_APDU, response_APDU *resp_APDU);

#endif
```

globals.h

```
/*********************************************************
* IAIK * IAIK * IAIK * IAIK * IAIK * IAIK * IAIK * IAIK *
*                                                       *
*   This is the implementation of a basic smart card *
*   OS supporting the T=1 protokoll.                          *
*   This version supports AES128 encryption and               *
*   decryption
             *
```

```
*                                                        *
**********************************************************
*****************************************
* File: globals.h                                    *
*****************************************/

/*--- Avoid including this file more than once ---*/
#ifndef _Globals
#define _Globals

/*
**============================================================================
**       INCLUDE FILES
**       Standard include files
**============================================================================
*/

/*
**============================================================================
**  3.      DECLARATIONS
**  3.1     Global constants
**======== ==================================================================
*/


/*--- TRUE / FALSE / NULL ---*/
#ifndef TRUE
#define TRUE            1
#endif

#ifndef FALSE
#define FALSE           0
#endif

#ifndef NULL
#define NULL            0
#endif

/*--- Return codes ---*/
#define OK            1
#define DATA          2
#define RTR           3

#define ERROR        -1
#define FULL         -2
#define EMPTY        -3
#define BUSY         -4

#define INPUT_BUFFER_SIZE 200

/*
**============================================================================
**  3.2     Global macros
**============================================================================
*/

/*
**============================================================================
**  3.3     Global type definitions
**============================================================================
*/

#define SIZE_BLOCK_AES          16
#define SIZE_KEY_AES            32
#define MAX_N_BLOCKS_AES   6
#define SW1_SUCCESS             159
#define SIZE_SW_RESPONSE   2;
```

```
typedef struct
{
    /*--- General ---*/
    unsigned char NAD;
    unsigned char PCB;
    unsigned char LEN;
    unsigned char CLA;
    unsigned char INS;
    unsigned char P1;
    unsigned char P2;
    unsigned char LC;
    int LE;
    unsigned char data_field[INPUT_BUFFER_SIZE];

} command_APDU;

typedef struct
{
    /*--- General ---*/
    unsigned char NAD;
    unsigned char PCB;
    unsigned char LEN;
    unsigned char SW1;
    unsigned char SW2;
    unsigned char LE;
    unsigned char data_field[INPUT_BUFFER_SIZE];

} response_APDU;
/*
**=============================================================================
**  3.4     Global variables (defined in some implementation file)
**=============================================================================
*/

/*
**=============================================================================
**  4.      GLOBAL FUNCTIONS (defined in some implementation file)
**=============================================================================
*/

#endif /* Match the re-definition guard */
```

T1_Comm_Lib.h

```
/*********************************************************
* IAIK * IAIK * IAIK * IAIK * IAIK * IAIK * IAIK * IAIK *
*                                                       *
*   This is the implementation of a basic smart card *
*   OS supporting the T=1 protokoll.                              *
*   This version supports AES128 encryption and                   *
*   decryption
             *
*                                                       *
*   DO NOT MODIFY!                                      *
*                                                       *
*********************************************************
*****************************************
* File: T1_functions.h                             *
* Version: 1.0                *
* Last change: 28.10.2005     *
* Author(s): Herbst Christoph *
*                                                                 *
*****************************************/
```

```
#ifndef T1_functions
#define T1_functions
#include "globals.h"


void send_ATR(void);
int request_extended_BWT(response_APDU *send_APDU , char extension_factor);
unsigned char receive_APDU(command_APDU *received_APDU);
void send_APDU(response_APDU *send_APDU);

#endif
```