Mid Sweden University
The Department of Information Technology and Media (ITM)
Author: Peter Edsbäcker
E-mail address: wmpeeds@netscape.net
Study program: B. Sc. in engineering, computer engineering, 180 ECTS
Examiner: Åke Malmberg, Mid Sweden University, ake.malmberg@miun.se
Tutor: Örjan Sterner, Mid Sweden University, orjan.sterner@miun.se
Scope: 31699 words inclusive of appendices
Date: 2010-06-12

Mittuniversitetet
MID SWEDEN UNIVERSITY

B. Sc. Thesis within
Computer Engineering C, 15 ECTS

SIM cards for cellular networks

An introduction to SIM card application development.

Peter Edsbäcker

Abstract

A SIM, *Subscriber Identity Module*, is the removable circuit board found in a modern cellular phone. It carries the network identity information and is a type of *smart card* which can also be found on payment cards (EMV), ID cards and so on. A smart card is basically a small computer, providing a safe and controlled execution environment. Historically smart card software was very hardware dependent and mostly developed by the manufacturers themselves. With the introduction of the open Java Card standard created by Sun Microsystems (Oracle) this was meant to change. However, information still remains scattered and is hard to obtain.

This paper is meant to serve both as an introduction to the field and also as a good foundation for future studies. It begins with a theoretical discussion about smart card hardware and software architectures, network standards in the context of SIM cards, typical applications, coming trends and technologies and ends off with an overview of the Java Card standard. The following section discusses the supplied example SIM card application coupled with an introduction how to use the *Gemalto Developer Suite* for application development and testing. The paper ends with an extensive appendix section going in depth about some of the more important subjects.

Keywords: SIM, Smart card, GSM, 3G, LTE, SIM Toolkit, Java Card, Global Platform, TPDU, APDU, Gemalto Developer Suite.

Acknowledgements

This bachelor's thesis was conducted during autumn 2010 and spring 2011. I would like to thank my teachers Örjan Sterner (for his dedication reading my many intermediate versions of this thesis) and Börje Hansson (for his inspiration). Both are based at Mid Sweden University. Also I would like to thank my old colleagues at Seamless Distribution AB [43]; it was through my work there that I came into contact with the field of SIM application programming, an interesting and to most developers, quite unknown, field of software engineering.

A big thank you goes to Google; whose excellent search engine helped me during the countless hours spent navigating the vast sea of Internet knowledge which serves as the basis of this thesis. Physical book references have mostly been avoided, the modern developer works and lives on the Internet in one way or another and books becomes obsolete very quickly. Another thank you goes to Gemalto N. V [2] for providing the (for evaluation purposes) free *Gemalto Developer Suite* that was used during development and testing of the example application provided with this thesis.

Contents

1 Introduction

1.1. Background and motivation

Historically smart card development has mainly been done directly by commercial companies (operators, smart card manufacturers and specialized consultant firms) using proprietary hardware and software solutions. With the introduction of the Java Card standard [14] this field of software development has finally been opened up to the general developer audience. New interesting applications and services for smart cards are now being developed by a broad range of companies whose solutions benefits from the security, integrity and robustness provided by a smart card. A well-known fact is that domain knowledge is the key factor that determines a projects' success or failure. However, in this field the necessary knowledge is not so easy to obtain.

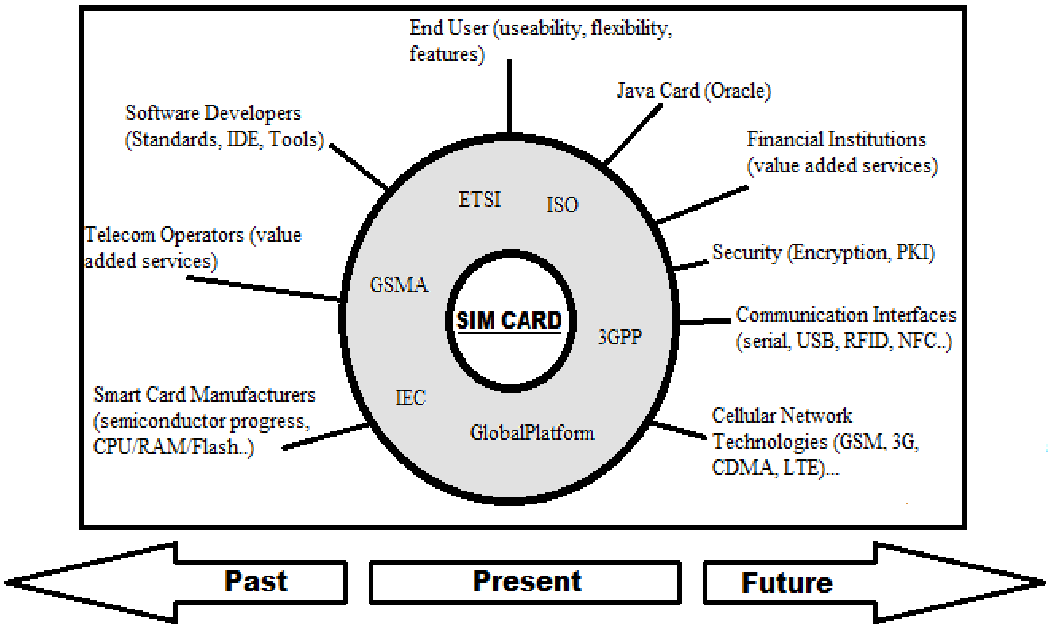

End User (useability, flexibility, features)
Java Card (Oracle)
Software Developers (Standards, IDE, Tools)
Financial Institutions (value added services)
ETSI
ISO
Telecom Operators (value added services)
Security (Encryption, PKI)
GSMA
SIM CARD
3GPP
Communication Interfaces (serial, USB, RFID, NFC..)
Smart Card Manufacturers (semiconductor progress, CPU/RAM/Flash..)
IEC
GlobalPlatform
Cellular Network Technologies (GSM, 3G, CDMA, LTE)...
Past
Present
Future

Figure 1: Some of the aspects that drive the SIM card evolution. The more important standardization organizations involved are shown in grey.

As seen in Figure 1 there are several interest groups and other factors driving the SIM card technology forwards. Unfortunately this diversity coupled with the inherent iterative process of progress reflects in the available information being very scattered and fragmented. Getting a good understanding of how all these technologies, standards, organizations and nomenclatures hang together is difficult, time-consuming, error prone and frankly, quite frustrating.

The author of this thesis has previously been working in the telecommunications field and there came into contact with development of applications for SIM cards. This experience gave the background motivation for writing this thesis. The work is an attempt to collect data from a broad range of sources, covering as many aspects as possible together with the authors own experiences into *one* document, thus forming a good platform for further studies in this interesting and quickly evolving field.

1.2. Overall aim

The primary goals of the thesis are to give the reader information about:
- Historical insight into the evolution of the smart cards.
- SIM cards in the context of cellular network evolution.
- Typical smart card applications.
- Current trends and technologies.
- The standards the smart card technology is based on.
- Smart card and terminal communications (i.e. between SIM card and handset).
- Internal structures of the smart card (hardware and software).
- Introduction to SIM card application development.

For the last point an example Java Card application was developed as part of the thesis work. This was done to give an insight how to program the card and also to give a very hands-on experience how to compile, upload and test the software since this is not trivial. The provided application implements user menu selection, text display, user input, dialing and sending SMS. It is implemented for Java Card version 2.1 (SIM version R99) which is still being the most common.

1.3. Scope

The thesis is meant as an introduction to the subject, scope obviously had to be limited since the field is huge and quickly branching out in many directions. Subjects deemed more important are covered with more depth, like the Java Card standard, Java Card/SIM application development, the APDU command set, smart card file system and the historical aspects. Others are only touched upon, like the authentication scheme, GSM network operations, smart card operating system internals, cryptographic functions, security and anti-tampering mechanisms, the Java Card class API and the new Java Card 3 standard.

1.4. Disposition

The paper consists of three main blocks, namely *theory*, *implementation* and *result/conclusion*. Chapter 2, 3 and 4 define the theoretical block. Chapter 2 explains some of the basic smart-card hardware and software architecture, current applications and new technologies. Chapter 3 defines more in depth what a SIM card is and its historical background as part of the evolving cellular network technologies. The theoretical block then ends with Chapter 4 that contains a discussion about the Java Card standard and Applet development.

Chapter 5 explains the process of developing an application, how the example application works and how to compile and test it. Chapter 6 discusses the end results and Chapter 7 the conclusions drawn from working with this thesis. This is followed by an extensive reference section (web links are included). Interested parties should particularly study the standards published by the ISO [1], ETSI [2], 3GPP [3] and Global Platform [16] organizations. Then comes a list of the abbreviations used throughout the document and by the field in general. The thesis ends with an extensive appendix section which goes into detail about historical background (A), smart card communications protocol (B), file architecture (C), SIM card command set (D), a list of smart card manufacturers (E) and the source code for the example application (F).

1.5. Audience

Primary audience for this thesis is:
- Application developers (preferably with a Java background) that want to get an introduction to SIM card programming.
- Scholars that wants the historical background, current state and new trends.
- Security engineers wanting basic insight of smart card security concepts.
- People with cellular network and/or operator background.
- Telecom business analysts interested in current and future solutions.

All readers should have basic knowledge about how cellular networks operate [3].

1.6. Method

The background material for this thesis has almost solely been collected through the Internet. Cross verification by using multiple sources has been attempted at a high degree. The bulk of the information has been assembled from information found in articles published by commercial companies, standardization organizations homepages and academic papers. The main reason for using the Internet as source is that it is kept up-to-date with the most recent trends and technologies and that it helps to find and link together the fragments needed to form a more complete picture of this complex and diverse field. The existing papers about this subject typically either go very deep into one specific topic without giving the necessary overview or are so shallow that they are useless for academic studies. Books covering this field are rare and quickly become outdated.

2 Smart cards

This chapter starts with an introduction to the concept of smart cards in general and then goes into more detail about the main subject of interest, the *SIM card*. The smart card technology is based upon standards that are created and maintained primarily by:
- ISO (International Standardization Organization) [1].
- IEC (International Engineering Consortium), in cooperation with ISO [4].
- ETSI (European Telecommunications Standards Organization) [2] ("TS xxx").
- 3GPP (3rd Generation Partnership Project) [3] in cooperation with ETSI.
- Global Platform group [16].

It is on these organizations homepages the standardization whitepapers, drafts and the like can be found and downloaded. ETSI and 3GPP papers are free but the ISO charges a fee for their services. References to various standards from these organizations are mentioned throughout the text. An overview is found below (Figure 2).

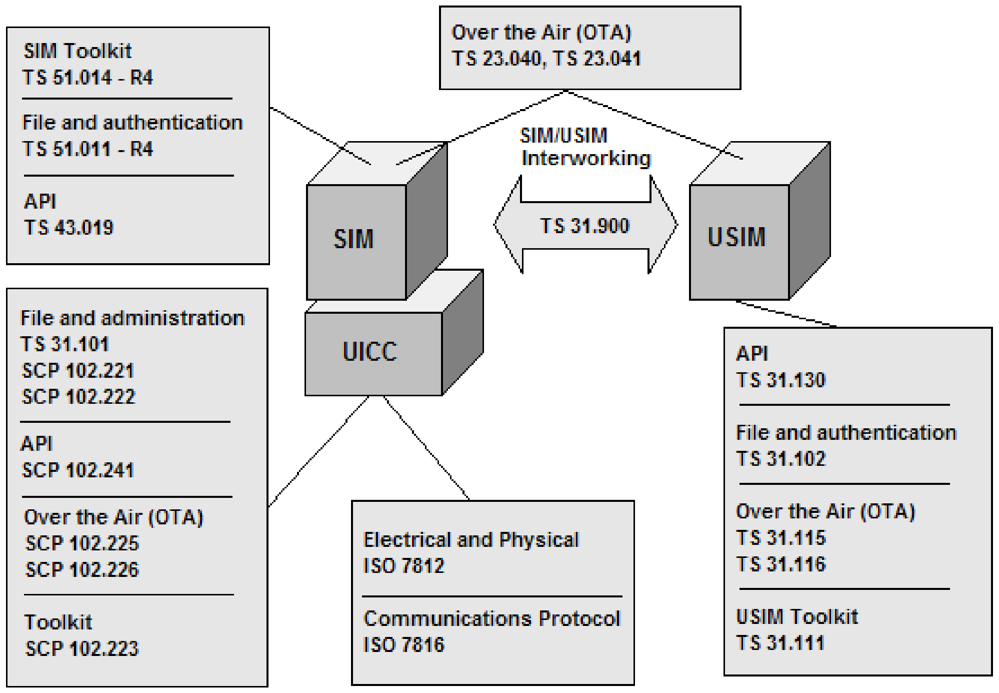


Figure 2: Overview of some of the SIM card standards shared between ETSI and 3GPP (TS and SCP above) [58].These relies on the foundation smart card standards, especially ISO 7812 and ISO 7816 [27]. SCP here stands for Smart Card Platform which basically is an adaption of the Java Card Platform for SIM cards.

2.1. The smart card hardware platform

Figure 3: Picture of a smart card, here, a SIM card for use in mobile phones, size is 25x15 mm.

A smart card comes in three physical sizes, as credit-card sized (credit cards), as the smaller 25x15mm (Figure 3) version found in mobile phones and as the new micro SIM format (12x15 mm). They cost anywhere from $1 up to over $40 depending on the hardware's capabilities.

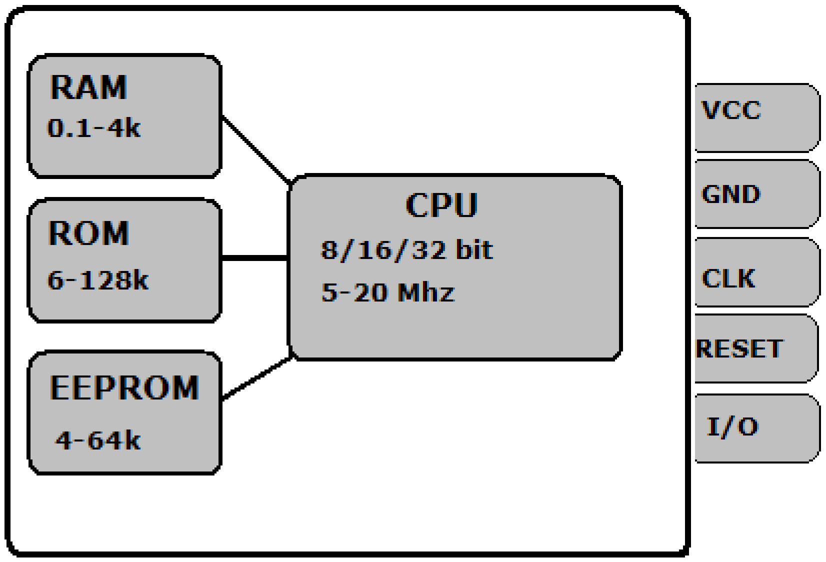


Figure 4: Smart card physical connections (standard pins) and its internal structure.

Figure 4 show typical specification ranges for a smart card found in cellular phones or on payment cards. There are more recent cards (Java Card 3) with much better hardware specifications but higher costs makes them still being rare. The hardware platform aspect of a modern smart card is called a Universal Integrated Circuit Card (UICC).

Features of smart card hardware

CPU: Historically 8-bit; typically a Motorola 6805 or Intel 8051 derivate, today the norm is 16-bit (Java Card 3 based generation use 32-bit RISC processors). The CPU can go into standby/sleep mode to save power where it uses much lower clock rates.

RAM: Size ranges from few hundred bytes to several megabytes; typically it is 1-4 kilobytes. There are often two types of RAM, *internal* that is used by the low level operating system and *external* (XRAM) that is used by the applications.

ROM: This contains the smart cards core operating system and support libraries. Sizes ranges from 6 kilobytes up to a few hundred kilobytes.

EEPROM/Flash: Programmable nonvolatile memory types which persists stored data also when not powered, but can be very slow to write (100-1000 times slower than RAM). The smart card's file system is stored in this kind of memory. Typically sizes are from 4 to 64k (up to 2 GB on recent cards [11]). The size of this memory type is often printed on the physical card itself ("32k" or similar).

Communication: Serial half-duplex 9600 bps (up to 115kbps for modern cards) communication interface/protocol between card and host. Half duplex means either the card or the host is sending data, not both simultaneously. Some recent cards also provide support for USB (up to 12Mb/s) and/or NFC (Near Field Communications) and RFID (Radio Frequency Identification).

Encryption and security: Most cards have custom encryption/signature (hash) verification hardware since the main CPU can be quite slow (especially for code running in a virtual machine). Usually it supports DES /3DES, RC4 but also more recently AES, RSA, DSA, COMP128 and Public Key Infrastructure (PKI) [44]. PKI private keys never leave the card. A hardware random generator is often present to provide more secure encryption key generation. The smart card has to provide protection against tampering (see Appendix C, section *hardware level security*).

2.1.1. The smart card hardware interface

A smart card has eight connecting pins according to ISO standards 7816-3/4 and ISO 7816-12[1].

Pin	Name	Description
1	VCC	Variable voltage, 1.8v, 3v, 5v (set after ATR).
2	Reset	Resets card and initiates the ATR (Answer-On-Reset) protocol, see ISO 7816-3 for details. The response contains the card's base capabilities and setup.
3	Clock	Typical clock rates are 5-20 Mhz. Clock is radically lower when the card is idle.
4	USB IC_DP	Optional USB/USB2 support[1].
5	Ground	Electrical ground
6	VPP	EEPROM programming (old cards). Now reserved for NFC SWP (Single Wire Protocol) to communicate with the NFC module.
7	Serial	Half-duplex serial I/O channel.
8	USB IC_DM	Optional USB/USB2 support[1].

Table 1: Smart card physical connection pins

[1]*The optional support for USB/USB2 is covered by ETSI document TS 102 600 and ISO 7816-12. There are also a new standard for contactless smart-cards that uses NFC/RFID [53].*

2.2. The smart card software platform

The first available smart cards had constrained memory space and could only provide very basic functionality. As we know, chips have quickly gotten smaller and more powerful and this has enabled the possibility to run (multiple) custom applications directly on the smart card. There are a number of reasons why someone would want to do this; the most important one is ***data security and integrity***.

The smart card is a closed environment, meaning you cannot upload software onto the card, nor access its file/data areas without unlocking it by using the correct authentication/encryption keys. Basically this means that both the applications and the data on the card can be fully controlled by the smart card manufacturer/supplier (for SIM - the operator).The encryption keys for data uploading and access are typically stored on the card during the manufacturing process and are normally unique for each smart card.

All smart cards issued before 1990 were programmed in assembly and/or "C" language [13]. The code was totally dependent on the underlying hardware and highly optimized to be able to fit in the constrained space. Porting an application between different card types (or even between manufacturers) was difficult and took lots of time. Bugs had to be avoided at all costs, replacing a smart card already deployed to the market was not possible (and still might not be today). This also meant application development typically were in the hands of the manufacturer (the product specialists), not the card issuers (for cellular networks; the operators).

This was clearly unacceptable and a solution was needed. It turned out to be virtualization. In the years 1990 to 1996 various researchers and manufacturers struggled to create a virtual machine that allowed the same application to be run on different vendor's hardware. However, it was not until the Java Card standard was introduced in 1996 that the industry really woke up (remember that Java was a really hot language back then, which was one of the key factors to Java Card's success).

The Java Card specification was originally developed by the manufacturer Schlumberger[2] together with Sun Microsystems (Oracle) [14]. It will be further detailed in section 2.3 but it basically is a virtual machine executing the applications in a controlled sandbox environment

2.2.1. Open platform / GlobalPlatform

Another standard was needed by the industry that described *how* to implement the new Java Card specification while still being hardware and operating system neutral. Standardized control over application lifecycle and security was also very much needed. The global payments provider *Visa* [23] pioneered this work 1998-1999 with its *Open Platform*[3] (notably this work was not initiated by a smart card manufacturer but a technology user). Their aim was to make a generic standard for all kinds of devices, even if their focus was to create a solution for secure payment systems.

As this project grew a stand-alone organization was needed and the GlobalPlatform [16] group was formed 1999. Its current members include Oracle, Ericsson, Gemalto, Nokia, VISA, American Express and many others. The group also collaborates with ETSI/3GPP for creating SIM card specifications. All Java Card enabled SIMs follow the GlobalPlatform standard, beginning with the SIM Card R99 (see section 3.1.3.1). It is also used for things like payment cards (EMV), identity cards, passage systems, biometric measurement devices, set top boxes, satellite receivers and many more.

Examples of features and functions provided by the JCRE and the Global Platform API that are defined by the GlobalPlatform standard:

- Application loader.
- Support for more than one application per card.
- Enhanced encryption algorithms (RSA, AES and ECC), encryption key management (and PKI).
- Life cycle management; possibility to upgrade/install/delete/lock applications after the smart card after it has been deployed (for SIM cards also over the air). The standard also specifies how the server-side (*host*) function for this is to be implemented.

- Security domains: PKI, encryption, decryption, digital signature generation and verification using keys visible only to Applets residing in the same domain.
- Global Services: cross-application calls using sharable methods.
- Requirements for the Applet firewall (memory data insulation between Applets residing in different domains/packages).
- Secure Channels Protocol (SCP) for encrypted communications between Applet and server.

One well known implementation of the Java Card and GlobalPlatform is IBM's *Java Card Open Platform* (JCOP) [17]. Note that the GlobalPlatform standard is not restricted to be used solely by Java Card, it is also supported by the MULTOS [24] operating system and others.

The GlobalPlatform still has problems specifying rules (i.e. using certificates or similar) for which applications that can call other application's methods [64] to make the behavior of post-issued Applets well defined and secure. This is one of the reasons Applets have to be certified (Common Criteria [59]).

[2] *Schlumberger's smart-card division later merged with a company called Gem-plus to form Gemalto [22]*.

[3] *The Open Platform standard is not to be confused with Open Card which is a host (PC)-side Java API to access a Java Card compliant smart card in an easy way (APDU processing helpers etc.).*

2.3. Java Card platform components

The Java Card virtual machine (JCVM) isolates the hardware specifics from the applications (called *Applets*). The JCVM is executing the Applets in a virtual CPU environment (see section 2.3.2). The applications make use of a class library called the *Java Card API* to perform its functions. The JCVM together with the Java Card API and the card's high level operating system functions is also known as the *Java Card Runtime Environment* (JCRE).

When we talk about the *Java Card Platform* (JCP) we mean the implementation of the features provided by the GlobalPlatform specification and those provided by the Java Card Runtime Environment (JCRE). A smart card implementing Java Card always follows one of the JCP versions. The terms *Java Card* and *Java Card Platform* are used interchangeably in most texts.

There are two families of JCP currently in use; the most common is the *Java Card 2* generation (versions 2.1.1., 2.2, 2.2.1 and 2.2.2). These are for use with more limited hardware. It is also the type found in most current SIM cards since it makes them cheap to produce. The other family, *Java Card 3*, was introduced in 2008. It uses an improved

JCRE and requires a 32-bit CPU (see section 2.7.5) which also makes it more expensive to produce. This family can be found on some modern 3G and LTE SIM cards.

Figure 5 shows the composition of the JCP, the details are given in the following sections.

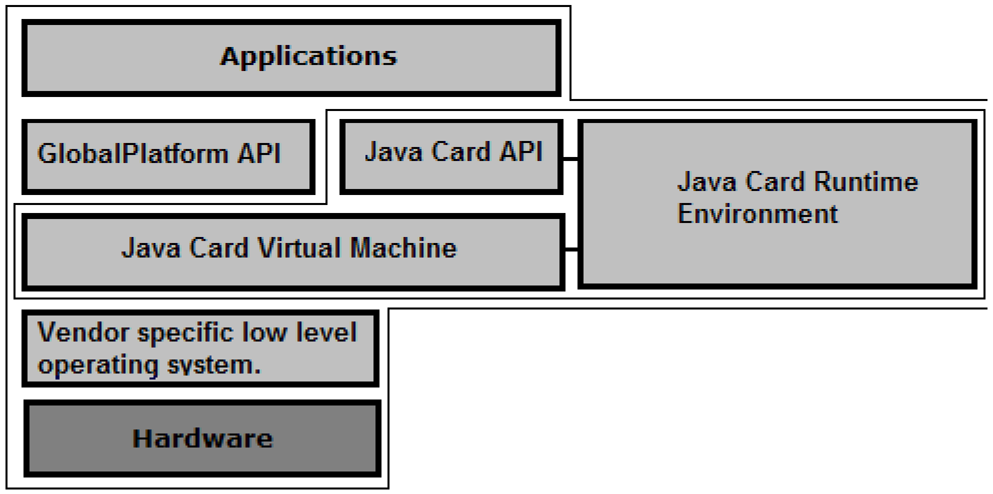

Applications
GlobalPlatform API
Java Card API
Java Card Runtime Environment
Java Card Virtual Machine
Vendor specific low level operating system.
Hardware

Figure 5 Software layers of a smart card based on the Java Card Platform. Optional additional items are various vendor and card issuer class libraries (APIs).

2.3.1. Low level operating system

This layer includes the base application management, serial communication, fast encryption libraries, real time kernel and so on. These functions are typically developed in assembly language (and/or “C”) by the smart card vendor /chip manufacturer for optimized performance and compact size.

2.3.2. Java Card virtual machine (JCVM)

The Java Card Virtual Machine (JCVM) sits on top of the low level operating system. This makes application solutions portable and increases both application security and integrity (sandbox execution). The JCVM is also a part of the Java Card Runtime Environment (JCRE) and is based on the standard Java Virtual Machine (JVM). Its byte-code is however more compact and limited. Today 184 op-codes are defined by the JCVM (v2.2) with over 200 being used by the JVM. Both virtualizes a stack based CPU (operations are done on stack operands). As such the byte-codes can be tricky to execute directly in an efficient way since most types of CPUs are register based.

Other types of smart card virtual machines, like *CAMILLE* [13], instead use register based op-codes (through a conversion process). CAMILLE even supports just-in-time (JIT) compilation to native machine code. The drawback is that register based op-codes takes more space. This forces a decision to be made between execution speed and the executable’s size. Today the balance falls in favor of the latter. Actually even *further* compression of the already quite compact JCVM byte codes is proposed [20].

The primary verification of the byte code for the JCVM is done during the conversion process (from standard JVM byte code to JCVM byte code). This step is done by the application loader in the JVM. However, the application loader found in the JCRE only has a very simple byte code verifier. It is as such sensitive to weaknesses that an attacker can exploit [18] [19]. Once the card has been deployed bugs found in the JCVM/JCRE cannot be fixed since they are placed in ROM. An Applet containing maliciously changed byte codes can thus be a problem since applications can today be downloaded after the card has been deployed (post-issuing). Thus commercial Applets usually have to be certified by some third party.

2.3.3. Java Card API (framework classes)

The Java Card API is the collection of base class libraries (known as *packages*) that the actual applications are built upon. These are defined primarily by the Java Card specification (base smart card libraries). Vendor specific extensions can often also be found on the card. The Java Card API classes are described in more detail in section 4.7.

2.3.4. GlobalPlatform API

This is the class libraries and the Open Platform foundation that is provided by the GlobalPlatform standard. They provide a neutral platform for application development (for example you can use the MULTOS platform instead of the JCRE). However, today applications need to use both the GlobalPlatform API and the Java Card API together. The GlobalPlatform API provides functionality to Applets as defined by section 2.2.1, most important are security domains, global services and the secure channels protocol.

2.3.5. Java Card Runtime Environment (JCRE)

This layer contains the "operating system" part of Java Card. Its functionality is defined by the Java Card Platform standard (which is influenced by GlobalPlatform). Note that the JCVM and Java Card API and sometimes even the applications themselves are counted as part of the JCRE.

Basic overview of the operating system functionality of the JCRE:
- Application (Applet) downloading management.
- Instance creation (of newly downloaded Applet).
- Activates applications (*select*).
- Deactivates applications (*deselect*).
- Manages the communication protocol (TPDU/APDU) and message distribution to and from Applets and the handset by invoking Applet class method *process*.
- Event management.
- Logical channel management (version 2.2.2 and forwards).
- Java Card Remote Method Invocation (RMI/MIDlet [63]).
- Contactless cards management.

2.3.6. Application layer (Applets)

This consists of the actual applications on the smart card, the *Applets*. Applets lives inside JCRE class packages. Some mandatory items have to exist, like the Applet's main class. The definition of "application" here actually means the executable's classes plus its configuration files and other parameters. For example the base GSM application consists both of on-card executable and the necessary network configuration parameters.

An (not pre-issued) Applet has the following life cycle [13]:

- *Produce and Load:* Upload binaries onto the card (managed by JCRE).
- *Initialize and Instantiate:* Initialize application instance (managed by JCRE).

First the application is *installed* by calling the static method *install*. This calls the constructor of the Applet, creating the Applet *instance;* it also *registers* the instance, which makes it selectable in the root menu.

- *Use:* Normal usage of the software is possible only after the first two steps have been performed. The use pattern as seen by the application is actually three phases (*select*, when application is selected in the menu or is going to become active for some reason, *process*, when the application is running and *deselect* when the application exits).
- *Revoke*. When the card is revoked for some reason it will be denied access to the network.

2.3.7. Smart card manufacturing, issuing and deployment

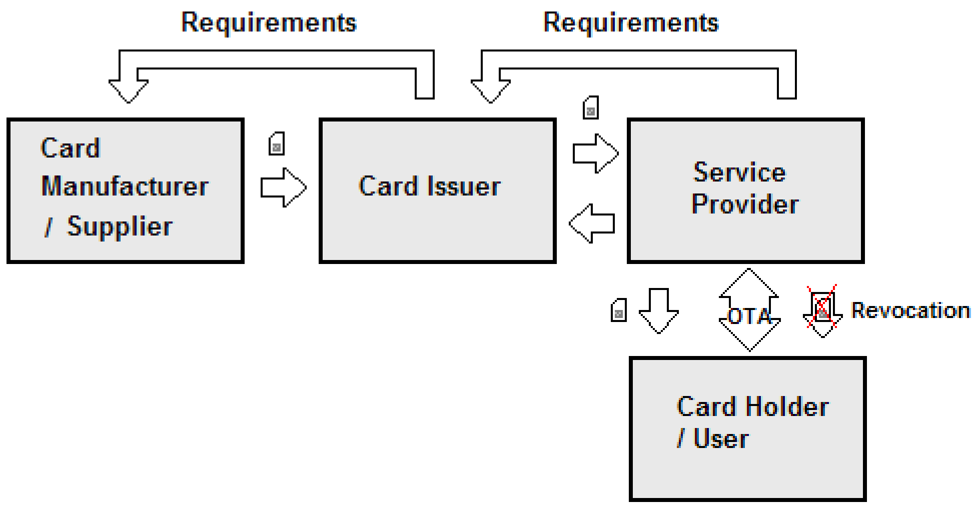


Figure 6: Simplified lifecycle diagram of a smart card. OTA stands for "Over the Air" application distribution and is only possible for SIM cards.

There are multiple parties (Figure 6) involved the smart card production life cycle [13] [60] which consist of five steps, starting at the chip manufacturer and ending at the user.

1. *Semiconductor Manufacturer:* Designs and creates the chips (examples; Infineon [21], Philips).
2. *Smart Card Supplier:* Creates the physical smart cards from the chips (example; Gemalto). Attaches additional things like NFC antennas.
3. *Smart Card Issuer*: Sets up the card, verifies standards compliance. Defines card hardware specifications. Sometimes they also distribute allowed memory usage and so on (issuing "mounting licenses") if there are multiple service providers involved.
4. *Service Provider:* The smart card is managed by a Service Provider. They typically can also revoke the card when needed and can deploy additional applications. Service providers works closely together with the *issuers* to control the content on the card. Examples of typical service providers are cellular network operators and banks (EMV). Other types of service providers just provide applications but do not otherwise control the lifecycle of the card.
5. *Card Holder / End User*: The actual user of the card

Note that the line drawn between Smart Card Issuer and Service Provider is not so strictly defined and the cooperation between them have anyways to be very strong. How responsibilities are split between them varies. Other parties involved are the ISO organization handling out AIDs (unique applet IDs) and various registration and certification bodies (depends on country/region and type of card).

2.4. Over the Air distribution of applications

Most modern SIM cards and phones allow for Over the Air (OTA) downloading [37] and upgrading of SIM applications (Applets). These are typically pushed out by the operator who wants all of their customers to have access to certain applications (like value added services) available in their phones. End customers can be allowed to request downloading of additional applications (like games and news) [46].

Great care has to be taken here to see that the customer really has enough space for this on the card. Actual distribution of applications was traditionally done through multiple SMS messages (see S@T, section 2.7.1) but is today done over TCP/IP (GPRS/3G etc.).

2.5. Typical commercial applications

There are a huge number of deployed applications not counting the necessary base applications (GSM application and the like). The following sections just represent a short list to give an idea about the possibilities.

2.5.1. Financial services

Recent forms of payment solutions for handsets uses Near Field Communication (NFC) to make shopping easier (also enabling services such as ticket purchasing, registering loyalty card usage and so on). Good examples of this technology are the *Google Wallet* [61] and the *Isis system* [62]. The owner of the handset just has to keep it in close distance to the resellers NFC receiver to complete a purchase, with the confirmation done on the handset (PIN or likewise), see Figure 7. This follows the same flow as using a traditional credit card but is simpler and also enables the user to keep track of purchases, credit and the like on the handset's display.

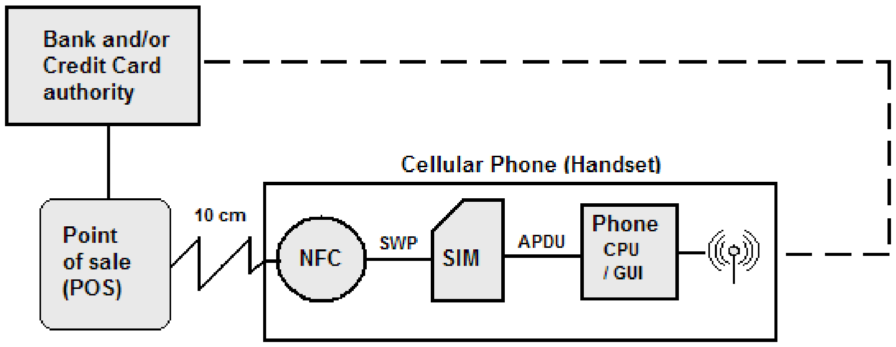


Figure 7: Simplified diagram of a NFC payment system. Phone/bank verification is optional.

The source of money is usually a credit card account or bank account tied to the user and his/her SIM card [54] but can also be the users call credit (air time) account at the operator[4] [43]. NFC-based solutions needs extensive security measures since wireless attacks can be done at a distance without the user being aware of it (section 2.7.2). Here the SIM card is used to perform secure authentication and encryption of data. It sits in the communication stream between the reseller's point of sale terminal and the operator/bank.

Another type of application for using air time in novel ways is used by street resellers doing “top-ups” of customer's call credit. The amount is transferred from the reseller's phone account. The reseller will take a charge for this service from the customer. This type of application is becoming very popular in developing markets such as the African region where the infrastructure is limited but cellular phones (handsets) are to be found everywhere [43]. If air time is allowed to be used as money this method can also be used to transfer money between bank accounts, for example an employer can pay their employees this way. This indirectly needs the involved network operator to become a financial institution with a whole new level of rules and regulations to follow.

[4] Not everyone in the lower income segment have a bank account, this is especially true in developing countries.

2.5.2. Electronic signatures

Most modern banks use some kind of security dongle to generate one-time codes for access to Internet banks. This kind of functionality fits perfectly for a SIM card application since people always have their phone with them [47].

2.5.3. Roaming control

Applications exists that can take care of roaming control when you are for example outside of the country and only want to allow to connect to certain operators (or none at all) because of cost reasons [51].

2.5.4. Value added services on a SIM

Examples of VAS services implemented on SIM cards:

- Missed calls alerting [49] [50].
- Caller number display (display name of caller even if not in your phonebook) [49].
- Advanced call forwarding.
- Multi-line (multiple phone number support on one SIM) [48].
- Automatic handset configuration (application configures phone by parsing incoming special SMS).
- Account top- up (air time refill) [43].
- Transfer of air time to someone else's account [43].
- Service ordering (games, news and so on).

There are a huge number of other operator supplied VAS service types available; however, most of them are implemented using USSD/WAP for user interaction or activated by sending a SMS with a specific content and as such fall outside the scope of the thesis.

2.6. New technologies

2.6.1. USAT (Extended S@T)

This is a standard to download Applets dynamically onto the smart card in order to provide an operator controlled *service portal* where they can provide various value added services to their customers (see section 2.6.4). S@T stands for "SIM alliance toolbox". It has been replaced by USAT which gives a WAP-like user experience trough an extended set of APDU commands defined by some of the more recent command classes (see section 3.4.1 and 3GPP TS 31.111).

2.6.2. Near field communications (NFC) and RFID

These wireless communications technologies are mainly used for credit-card-on-a-phone solutions today (section 2.6.1). However, it can also be used for peer-to-peer communications (known here as *Machine2Machine* [11]) when the devices are physically close. This works much like Bluetooth but is simpler since there is no need for device pairing. NFC can for example be used as a simple way to download new applications to the phone (and/or SIM card) or to provide various financial services (section 2.6.1)

There are different NFC standards depending on the reachable distance (1cm, 10cm and up to 1 meter). The communication speeds are in the range 106 to 848 kbps. A NFC unit usually resides inside the phone (however, there are SIM cards with NFC on-board) with a special secure serial line connecting it to the SIM card. NFC also enables RFID (Radio-frequency Identification) applications [55]. The SIM card is an excellent service authenticator and might enable the phone to act as a door key, enable access to other restricted resources (using PIN or do a biometrics check of some sort) or to identify merchandise using RFID tags.

In conjunction with credit card usage RFID/NFC could be used for things like enabling a reseller to check that the phone owner and his/her credit card are physically close to limit fraud. This is however a good example that using RFID technology might come at a cost of personal privacy and integrity due to the possibility to track individuals [56]. These wireless technologies are also a welcome target for hackers since attacks can be made at a distance with little chance of detection [57].

Wireless protocols are governed by the following ISO/IEC standards:
- 14443: Contactless cards
- 18092: NFC
- 15693: RFID
- 21481: Selects which wireless communication mode to use (NFC, RFID etc.)

2.6.3. Virtual Sim card

This technology is aimed to create a fully software based SIM card that is executed by the phones operating system, bypassing the need of a physical SIM card and enabling multiple phone numbers for one physical handset. Not surprisingly Apple Inc. is one of the driving forces behind this movement [52] and a group has been assembled by the GSMA (GSM Association) to create a standard [6]. Very high security is demanded if such a virtual SIM is to be possible to move between phones and to avoid data duplication and tampering.

2.6.4. USB and USB-2 support

The new smart card USB/USB2 support is covered by ETSI TS 102 600 and ISO 7816-12. The standard currently supports USB speeds up to 12Mb/sec. Together with the new functionalities provided by Java Card 3 USB support opens up for a secure web-server-on a card or using the smart card as a stream encryption device. In conjunction with recent smart cards having up to 2 GB of storage USB enables the smart card to be used as a flash disk for storage of pictures and video-clips [11].

2.6.5. Java Card 3 and GlobalPlatform 3 base technologies

Java Card 3 enables more Java-like applications to be written and has support for multithreading and multitasking, garbage collection and many of the usual business objects found in Java SE.

Some of the new features introduced are:
- TCP/IP over USB.
- Smart Card web server using HTML 1.1/HTTPS (SCWS, enables Applets to have a HTML based GUI on the phone). It also provides Servlet support.

See section 4.3.2 for some more information about Java Card 3standard.
With the introduction of LTE networks (and late 3G) a new generation of SIM cards is being deployed that are Java Card 3 / GlobalPlatform 3 based.

2.6.6. Secure VoIP, Mobile broadcast TV

The *GlobalPlatform 3* standard and its more powerful hardware enables enough power to do real-time encryption and decryption of streamed data through the SIM card. Example areas of such usage are secure VoIP and Mobile broadcast TV [11].

3 SIM Cards and cellular networks

When the GSM standard was proposed there was an obvious need for a strong user/network authorization. This meant a telephone number was to be closely tied to a subscriber account in the operator's network and at the same time making it very hard for someone to copy the information (since this might enable debiting calls on somebody else's account). One way to solve this would mean putting the phone number, necessary encryption keys and the like inside the physical phone itself. This was the method used in older American CDMAone-based networks.

However, it meant that the user got one phone number for each physical phone, making replacement a big problem. In order to avoid this in the GSM networks, the authentication and user identity functionality was placed on a removable smart card. This smart card type was called a *Subscriber Identity Module* (SIM). The smart card command set as defined by the ISO standard was extended to make it possible for the SIM to perform user interaction. Examples of such commands are the ability to display text on the phone's display, get user input and sending/receiving SMS (see Appendix D).

3.1. The historical evolution of the cellular phone standards

Before further discussing the SIM cards we need a little historical background about the cellular network types. It was the technology advancement of these standards that directly has driven the evolution of the SIM cards (both software and hardware-wise). In the recent years we have seen how the European GSM and the American CDMA standards have merged into the 3G (UMTS) and 4G (LTE). These form the global standard platform for future cellular networks around the world.

3.1.1. GSM

Global System for Mobile Communications (GSM) [6] is still the most common worldwide mobile phone standard. It uses full digital signaling in respect to its predecessors which used a mix of analogue and digital signaling (NMT and others) and uses *Time Division Multiple Access* (TDMA) as a radio carrier/protocol. It was from the beginning used in Europe. GSM is considered to be a *second generation* (2G) mobile communications standard.

Later *General Packet Radio Service* (GPRS) added data packet handling to GSM networks (for consumers it enabled things like MMS, WAP and IP). The GSM, GPRS (and Edge) standards are controlled by ETSIs *Smart Card Platform* (SCP) committee. Private promoters of these standards are the GSM Association (GSMA) [5]; a big group of phone operators and related companies. GSMA aims to spread GSM related technologies throughout the world. It was the GSM standard that pioneered the concept of using smart cards in cellular phones.

3.1.2. CDMAone/CDMA2000

CDMAone is the most common American 2G standard which is equivalent to GSM. It was also commonly used in Japan. It uses *Code Division Multiple Access* (CDMA) as radio carrier/protocol. Originally it did not require a SIM; the phone number was thus directly associated with the physical phone. CDMAone has since been superseded by the third-generation (3G) standard *CDMA2000*. Both of these standards are controlled by the 3GPP[5] [16] group. CDMA2000 introduced the R-UIM (Removable User Identity Module) which is based on the GSM SIM.

Internally the R-UIM runs at least two applications on the same card:

- A CSIM application (CDMA subscriber identity module) for CDMA networks (contains network parameters, phone number, SMS storage etc.)
- A SIM (GSM SIM) application for GSM networks.

As such, the R-UIM card can be used in both CDMA2000 and GSM networks. However, it has later been superseded by cards following the more modern UICC smart card standard (more specifically, the *U-SIM* family).

[5] *Do not confuse 3GPP2 (mostly American/Japanese interests) with 3GPP*.

3.1.3. UMTS and LTE

Universal Mobile Telecommunications Systems (UMTS) [12] defines the by far most common 3G standard. It uses improved CDMA (W-CDMA etc.) as radio carrier/protocol with the GSM standard as its foundation/fallback. The standard is controlled by 3GPP; note that the ETSI group is a (major) part of the 3GPP project as are the AIS (Alliance for telecommunications Industry Solutions) which in turn are certified by the ANSI (American National Standard Institute) [8] [9].

Other important group members are Chinese and Japanese organizations with the same function in their part of the world. Basically this means that the 3GPP group is creating *the* de-facto world-wide standards for cellular networks for third (and future) generations. The major 4G standard *Long Time Evolution* (LTE) [40] is an extension of UMTS and also supersedes CDMA2000. It is thus the major worldwide 4G standard. Internally LTE uses IP (Internet Protocol) everywhere in the network. It provides very high bandwidths, multimedia support and automatically falls back to GSM/3G UMTS/CDMA2000 etc. in case there is no LTE carrier available at the current location.

3.1.3.1. GSM, UMTS 3G and LTE 4G version history

The network standards GSM, UMTS and LTE are based upon a progressive evolution. There have been several releases during the years [39].

Year	Release	Notable new features	SIM card version
1997-1998	R96, R97, R98	GSM	As per 3GPP/GSM standards 11.11 and 11.14. These are *not* Java Cards.
1998	R98	EDGE, GPRS	
2000	R99	UMTS (3G)	SIM Card R99 (first Java Card).
2001	R4	Core IP network	
2002	R5	Multimedia, HDSPA (Turbo-3G, also known as “3.5G”).	SIM Card R5 USIM Card R5
2004	R6	W-LAN, Multimedia enhancements etc.	USIM Card R6
2007	R7	Voice over IP, HDSPA+ and more	
2008	R8	LTE (4G).	SIM Card R8 (Extended OTA support). Contactless SIMs (NFC).
2009	R9	Wi-Max interoperability	(UICC for 3G/LTE comes with Java Card 3.x ,USB and IP [11])
2011 Q1	R10	LTE Advanced (full LTE, 1Gbit/s)	
2012?	R11	Advanced IP interoperability (IP between operators)	

Table 2: Cellular network generations, as seen some releases introduced new SIM versions.

3.2. SIM card configurations

3.2.1. SIM network applications

There is always one or more applications on the SIM card that gives the user access to the cellular network (authorization and so on).

GSM / SIM

This is the application that connects the phone to the GSM network. It only authenticates the phone with the network, not vice versa (section 3.3.1).

CSIM

CSIM are used in CDMA2000 networks and contains similar network parameters as the GSM application found on a SIM card. They used to run on an R-UIM type card but now run on a UICC.

USIM

The *Universal Subscriber Identity Module* (USIM) application is used in UMTS 3G/4G Networks. It authenticates both the user with the network and network with the user and thus incorporates better authentication mechanisms than GSM (also uses longer keys, better encryption etc.). Apart from 3G/LTE extensions it contains the GSM (2G) network parameters/application as-well, since GSM is used as a fallback if there is no 3G/LTE coverage in the current area.

ISIM

The *IP multimedia Services Identity Module* (ISIM) application might be found on modern cards for 3G/LTE which are based on GlobalPlatform version 3. The application gives access to IP Management Sub-system (DRM) [11] services used for things like streamed video.

3.2.2. SIM card hardware types

SIM card

The old type of smart card that has been used in GSM (2G) networks. It has been superseded by the UICC card. The usage of the term “SIM card” today has been expanded to encompass the whole plethora of card types.

R-UIM cards

These contain all three (GSM+CSIM+USIM) network applications and are usable in all the corresponding networks types. They were first released for use with CDMA2000 networks (the American 3G standard). Old style R-UIM has been superseded by the

UICC standard but the name is still being used for UICC cards with all three mentioned applications present.

UICC cards

The *Universal Integrated Circuit Card* (UICC) is the smart card type which all modern SIM cards are based on. It can host all applications of section 3.2.1 and also implements Java Card. Today the vast majority of cards are based on this smart card type. Older non UICC-based cards might not even be allowed in operator networks due to security concerns.

3.2.3. SIM card generations

Each generation of card implements a new set of Java Card and application development features as described below. They also define the new network configurations and applications related to the revision of the network associated with the SIM card type.

SIM Card type and release	Specifications and features for on-board application development
SIM Card R99	Open Platform 2.0.1, Java Card 2.1.1. First SIM version that has Java Card on-board.
SIM Card R5	Java Card 2.1.
USIM Card R5	Java Card 2.2.1, Advanced phonebook, higher serial communication speeds.
USIM Card R6	GlobalPlatform 2.1.1 (2003) and Java Card 2.2.1, ISIM application.
R-UIM Card	GlobalPlatform 2.1 and Java Card 2.2.1.
(LTE/3G cards)	GlobalPlatform 3.x and Java Card 3.x (recent).

Table 3: SIM card types.

3.3. GSM network parameters

All SIM cards contain network parameters and encryption keys at well-defined places in the file system (see Appendix C). The file system also contains the network base application(s), these applications are responsible for communicating and authenticating with the operators network (handshaking encryption keys and so on).
Most of these parameters are defined when the SIM is manufactured because they need root user access to the card or reside in ROM).

A short list of necessary files (EFs) needed to use a SIM card in the network, see GSM 11.11 [3]:

- The phone number (called MSISDN).
- SIM's network identity; the IMSI (International Mobile Subscriber Identity).
- A unique SIM serial number (called SSN and/or ICC-ID [42]).
- A 128 bit network authentication/signing key[6] (called *Ki*) to authorize the phone to the network (see section 3.3.1).
- PIN code (entered by user when user starts the phone).

- PUK code(s), entered to unlock locked phone because of too many PIN retries.
- Network/operator specific parameters.
- Smart card access authentication keys, these are needed to allow applications to be downloaded onto the SIM and to access/change certain areas of the SIM file system (such as changing the MSISDN and other operator/network specific settings). Typically the authentication keys are using 3DES encryption.
- Storage for SMS messages.
- Storage for a phone-book (at least 250 items on modern cards).

Examples of information stored/accessed by the cellular phone or pushed there by the network:

- Current network state and cell information (LAI/TMSI).
- The number to the SMSC (SMS provider [41]).

[6] *GSM uses the A5/1 or A5/2 encryption standards which in the latter years have been proven to be (rather) easily broken. The encryption algorithm has been improved for UMTS/3G networks with A5/3 [35]. 3G also uses a 128 bit session key for data encryption whereas GSM uses a 64 bit key.*

3.3.1. Authentication of a cellular phone with the GSM network

The authentication process is done by the SIM card (more specifically by the GSM application running on the card) and is based on a challenge-response pattern [10]. The details falls outside the scope of the thesis but the basics are shown below in Figure 8.

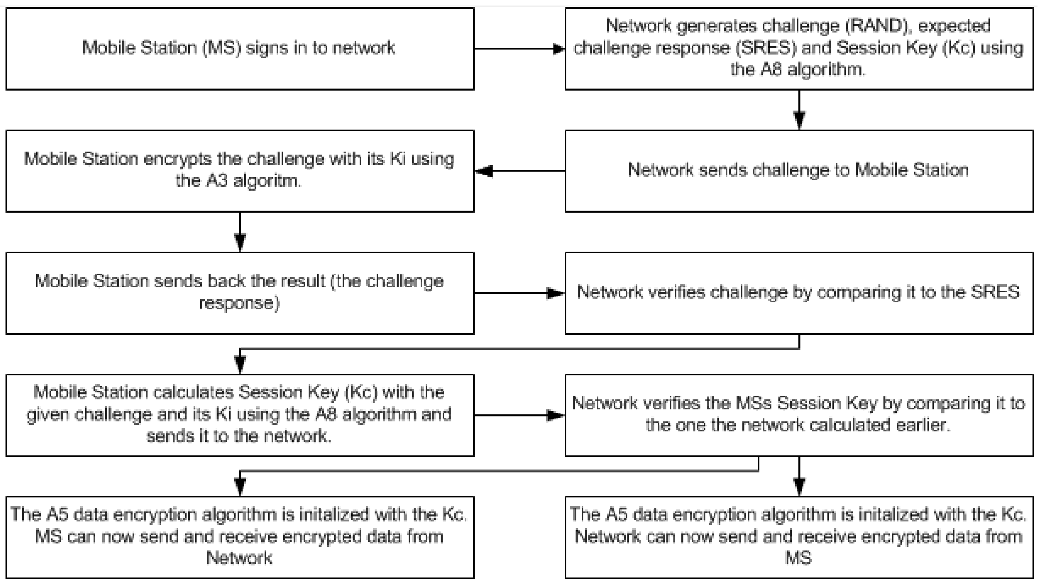

Mobile Station (MS) signs in to network
Network generates challenge (RAND), expected challenge response (SRES) and Session Key (Kc) using the A8 algorithm.
Mobile Station encrypts the challenge with its Ki using the A3 algoritm.
Network sends challenge to Mobile Station
Mobile Station sends back the result (the challenge response)
Network verifies challenge by comparing it to the SRES
Mobile Station calculates Session Key (Kc) with the given challenge and its Ki using the A8 algorithm and sends it to the network.
Network verifies the MSs Session Key by comparing it to the one the network calculated earlier.
The A5 data encryption algorithm is initalized with the Kc. MS can now send and receive encrypted data from Network
The A5 data encryption algorithm is initalized with the Kc. Network can now send and receive encrypted data from MS

Figure 8: Simplified diagram over GSM-based handset authentication with the network.

A handset (in literature also called ME; Mobile Equipment or MS; Mobile Station) contacts its network. The network then contacts the home location register (HLR) for the operator in question. The HLR looks up the users account by the phone's MSISDN. In the account the corresponding *Ki* is stored (among other things). The HLR then generates a set of keys (based on MSISDN/IMSI and the Ki). These are sent back as a *challenge token* (a question of type "return to me the result if you encrypt the given random data with your key").

The SIM card sends back its *signed response* to the network which verifies it with what the HLR calculated. If it is accepted the SIM generates a 64-bit session key called *Kc* (through the A8 algorithm). This is based on the Ki and the received challenge token. This session key is used to initialize the A5 encryption algorithm (on both sides) that is hence used during further communication.

GSM suffers from the limitation that it is only the handset that is verified by the network, not the other way around. An imposter (needing a fake base station) could thus try to figure out the handset's Ki by issuing challenges and looking at the response it gets back.

3.4. Communication between SIM and Handset

Basically the communication between the SIM and the handset (ME) is done over a serial communications line as defined by the smart card ISO standard 7816.
There are four protocols in use to transfer data (however TPDU and APDU protocols are overlapping).

- TPDU (Transmission protocol data unit)
This is the base smart card standard protocol for sending and receiving data over the serial connection. A modern SIM supports several (four or more) ongoing transactions at one time, these are managed by something called *channels* (See Appendix C).

- APDU (Application protocol data unit)
APDU is a transactional higher level protocol which piggybacks on-top of TPDU to save space (not following the OSI layered style). The APDU command-set define the SIM/ME interaction. Normally a smart card is a slave unit, here meaning it is the ME that sends commands to the SIM. But the SIM cards are highly active entities so a scheme was implemented to work around this limitation called *proactive commands.* Basically the ME polls the SIM regularly; the SIM then responds what it wants to be done (if anything). The ME processes the request and makes a transaction back to the SIM telling it the result. This can be made asynchronously (the SIM might continue other processing while waiting for the response). For details see Appendix B and C.

- JCRMI (Java Card RMI)
RMI calls can be made from a Java based host (client) to the smart card (server). The RMI data is encapsulated inside APDU commands and is handled by the RMIService class [45] on the smart card side (JCRMIMIDlet).

- Web Services (Java Card 3 and later *only*)
Java Card 3 introduces modern SOAP/REST style communication over TCP/IP which makes life much easier for developers. Cards that implement this standard typically also have support for USB communication; HTML is very unfit for transfer over the standard serial line.

3.4.1. Applet-Handset interaction

Something that an Applet developer *must* be aware of is that not all handsets implement all APDU commands defined in the standard (ETSI TS 102 223). You can fortunately query the handset about its capabilities by issuing the *Terminal Profile* command (see Appendix D). Based on this you might have to limit the functionality of your application or even in the worst case show an error message that the handset in question is not supported. Modern smart phones implement most of the commands but older GSM handsets can be a nightmare. Even if the response says that the handset implements a certain feature or command it might not fully adhere to the commands specification.

Most of the older base APDU commands have a corresponding bit in the Terminal Profile response telling if it is implemented or not. More recent APDU commands are instead associated with a *command class* (denoted by one or two lowercase letters). These also have corresponding bit fields in the response. The ETSI standard describing the APDU commands have been released in several versions as new classes and commands are introduced. For example APDU support for TCP/IP did not exist in the initial release of ETSI TS 102 223.

Major APDU command class categories

- Class “a “: Multiple SIM card support in same ME (communication between SIMs).
- Class “b”: Access to handset GSM modem (Hayes commands).
- Class “c”: Internet access (for example open browser with specified URL).
- Class “d”: Soft key support (SELECT ITEM, SET UP MENU).
- Class “e”: Card channels and bearer independent protocol support (server access).
- Class “f”: Services (USAT service support).
- Class “g”: Local information (country code, date, time, network cell information etc.)
- Class “h”: Multimedia support.
- Class “i”: Frames (handset can create a split screen GUI).
- Class “j”: MMS support.
- Class “k”: SIM card can initiate start of an application on the handset.
- Class “l”: Interface activation support (NFC/RFID module access, see ETSI TS 102 613).
- Class “m”: Host Controller Interface connectivity (typically wireless host communications).
- Class “n”: Geographic support (GPS).
- Class “o”: Broadcast network information.

4 Developing SIM applications

4.1. Obtaining SIM cards for development purposes

You can order blank SIM cards from SIM manufacturers for application development purposes. At the end of this document there is a list of vendors that can provide development SIM cards. A blank Java Card 2.x compatible SIM just costs a few dollars in bigger quantities. Such SIMs obviously does not have any network keys on them, but the necessary GSM file structure is already in place. You can later set these keys to enable access to network (the author has only done this for a test network). A normal SIM is configured already during manufacturing.

If you want to use your SIM-card applications in a real network you must first register your company with the ISO organization (see ISO standard 7816-5) so you get a unique *Registered Application Provider* (RID) prefix code. This code is the first part of the *Application Identifier* (AID) that all smart card applications must have. It must be unique world-wide. The RID ranges are assigned by the Copenhagen Telephone Company Ltd. (KTAS/Tele Denmark), which is also the ISO/IEC 7816-5 Registration Authority[7].

[7]*The address is Teglholmsgade 1, DK-1790, Copenhagen, V, Denmark. The application has first to be approved by your national ISO body, applying for a RID costs approximately $500 (2010)*.

4.2. Development environments

We will here concentrate on SIM application development, however, it is basically the same as developing for any GlobalPlatform compliant smart card. SIM cards have an extended command set for SIM/Phone and Network communication. The development environment for SIM card applications is called "SIM Toolkit" (STK for short).

There are basically two viable options for SIM application development, namely Native STK (vendor specific, typically programmed in Assembly and/or in "C) or Java Card/GlobalPlatform. Today the former is made obsolete since the R99 SIM standard and forwards require the latter. Native STK programming is however still being used for other types of more restricted smart cards (like RFID tags).

4.3. Java Card v2.x Applet basics

Since this Java Card family is what is to be found on almost all current SIM cards we restrict the discussion to what it provides. Each application is written as an *Applet*. An Applet and its surrounding helper classes are placed in a *Package*. It is also possible to form a class library by creating a package which can then be referenced by other applications which saves space. Each package has private (protected) memory areas and files isolated by the JCRE's application firewall.

General guidelines and rules for Applets based on Java Card 2.x:

- The application (Applet constructor) is initialized *once*, when the application is installed. Most application data, *including* objects are created during this process. They will be persistent and thus remains intact even when the application is not executing.
- Only one Applet can be executing at one time (*select, process, deselect pattern*).
- Two Applet accessible memory heaps exists, one for persistent data in FLASH/EEPROM and one for non-persistent in RAM (a very limited resource, a few hundred bytes including stack).
- Beware of small stack space; keep track of total data on stack at deepest nesting level (which is probably when you invoke some APDU command deep inside your code). Your application can crash unexpectedly. *Carefully* define the stack size you will need.
- Most application data will thus be persistent (remain at power-off). Code your logic carefully so it handles power off (called *tear*) at any time, committed data will remain when the application is restarted. Treat persistent application data like it is residing in a database.
- Transaction support is provided. Either the whole transaction (data update) is committed to persistent memory or nothing at all.
- Even though the application data might be persistent it is lost if the Applet is upgraded. Store your data in the file system.
- Work buffers must be allocated in transient (RAM) memory since persistent memory cells might stop working after 100,000 write cycles and also has *very* slow write access.
- The global work buffers are usually allocated once (in the constructor). Sharing buffers between methods is horribly error prone. It is typically necessary since you usually cannot allocate/free memory on the fly (C-style).
- Usually there is no garbage collection support.
- Cryptographic support included such as DES, 3DES, RSA, MD5, SHA, AES and others and can have quite a decent speed due to use of special hardware and/or low level code implementations.
- Application data can be shared between Applets (shared objects are governed by the *Applet Firewall*). By default Applet data/files are kept *strictly* separated.
- Not all handsets implement all APDU commands (or only partially), see section 5.9.2.2.
- Be careful so you absolutely do not take more than 5 second of calculations in one go (waiting ADPUs are fine). Other applications, like the GSM application (which need to generate network keys) have to be able to respond in time to events (timer, network and so on).Very strange things can happen if you do not adhere to this rule (for example network disconnects).
- Most cards have no JIT (Just in time compilation to native machine code). This means code execution can be very slow. Heavy calculations (like custom encryption code) should be avoided if you do not know exactly which cards you will deploy on so you can know the timing constraints. Otherwise you might violate the above rule.
- Use Java Card API provided helper classes whenever possible. They can help you save precious space in the executable.

4.3.1. Java Card 1.x/2.x limitations (with respect to standard Java)

- Applications are 16 bit (machine integers are 16-bit; you have to typecast to “short” everywhere). 32 bit integer support is optional and rare. There is no “char” data type.
- No threads, only one application can be running at one time.
- Typically no garbage collection.
- No class finalizer support (since the class will probably never be garbage collected)
- No dynamic library loading.
- No reflection.
- Arrays can only be one-dimensional.
- No string class support.
- No floating point support (float, double types are missing).
- Security is managed implicitly by JVM and not trough a configurable manager class.
- Persistence handling is very different from standard Java.
- Very restricted RAM (volatile) memory space.
- Memory space is restricted to 32k per package.
- Limited call depth (take care to limit nesting) and local variable sizes (RAM space).
- Constants should be defined “static final” to conserve space.

4.3.2. Java Card 3.x “connected” edition limitations and differences

- Still no float / double types.
- Has garbage collection.
- Has string type, 32 bit integers, multi-threading.
- Most standard base Java libraries are included.
- Loading and linking of class files are done on the card.
- SSL, TCP/IP over USB, NFC and so on.
- Contains a WEB server with Servlet and HTML support using HTTP 1.1/ HTTPS.
- Has a special virtual machine on board to run legacy Java Card 2.x applications (“classic edition”).

4.4. Applet intra-vendor compatibility and virtual machine quality

The features and possibilities with the Java Card (v2.x) were all obviously very promising, but still today the behavior can differ between vendors even if card certification is much better now. Some features are partially implemented or not at all. When you develop a SIM application, be sure that you will encounter bugs or unexpected behavior on some SIMs, especially if they are a few years old.

Most Java Cards of generation 2.x does not implement garbage collection of memory and all have very limited RAM space. This means that any “normal” Java coding is not possible and hence no real object oriented programming (OOP) is possible either. The coding breaks down to look like “C” code with lots of constant variables, static methods and other ugliness with lots of hacks to conserve binary code and memory space.

Some vendors do not use the JCVM binary code (i.e. CAP files) directly. Instead they use special binary file formats, which mean you have to recompile your Applet using their tools. You often have to carefully specify the correct SIM revision target (R99, USIM etc.), since the underlying APIs might be different. Java Card 3.x uses more standardized file formats and a better JCVM (with garbage collection and so on) which is quite promising. Still it does not provide the full functionality and power of the Java language.

4.5. Applet execution speed

The speed of the operating system and the virtual machines they execute varies wildly between vendors and Java Card generations. Note that writing and reading EEPROM/Flash is magnitudes slower than using on-card RAM. The type of memory that is used in the smart card also affects speed heavily.

Fortunately technology evolves and this should become much better by time, especially with the evolution of CPU technology, flash memory evolution and the introduction of Java Card 3.0 which puts much higher demands on the quality of the underlying kernel/operating system and the hardware itself.

4.6. Java card memory management and memory types

The pre 3.x generation SIM card has very little available on-board RAM memory, often only a few hundred bytes forcing you to place data in EEPROM or FLASH memory (nonvolatile/persistent storage) which can have up to 1000 times slower write cycles than RAM. Reading speed is approximately the same. Note that a flash cell typically has a life-time of approximately 100,000 write cycles, which means the operating system on the card must (should) keep track of potentially burned-out memory blocks. On newer cards there is memory block "virtualization" so writes can be spread out evenly throughout the memory (like how flash memory is used in SSDs). Flash memory has the peculiar way of working; it is only possible to *clear* individual bits, when a block (1-64k) is initialized (*flashing*; which is quite slow) all bits in the block are set to binary ones.

When you use the "new" operator in Java the object is created in *persistent* storage which is quite different from the behavior that developers are used to. A very important fact about persistent objects is to remember that the contents of data/objects residing inside it will *remain* when the application is restarted. This is true even during power-off/on cycles. Not all cards employ garbage collection, which means it might be impossible to de-allocate data areas (without deleting the whole Applet).

This means work buffers and the like are typically allocated ONCE in the applications initialization phase (the only time the Applet constructor is called). If you plan to be able to upgrade your application afterwards you should store data in the card's file system since the Applet's persistent data is lost during the upgrade process. When you write to a field in a persistent object the write is *atomic*, this means it either succeeds fully or reverts back to its previous value if the card loses power during the write operation. This is a very important feature to be able to retain data integrity on the card. There are library helper methods that provides support for transactions when multiple values are to be updated in persistent memory (either all are written or none at all).

4.6.1. Transient memory

It is possible to create transient byte buffers in RAM with special library methods. Being transient means that the data in the buffer is lost (zeroed) when the Applet is started or stopped. Be very careful not to use too much transient space since this might also limit your application's stack space. In Java Card 2.1 no true objects can be created in transient memory, only primitive types (integers etc.), arrays and arrays to object references are allowed. This restriction was somewhat lifted with Java Card 2.2.

A transient buffer is an area in the RAM that is reserved for your application. You need to free it *explicitly* with a call to the memory sub-system; the actual memory area will be kept reserved for your application and is not freed just because you application stops or the card is powered off/on. There are two basic types of transient memory:

Transient Deselect
Memory area is cleared when your Applet is de-selected (exited) or the card loses power.
Transient Reset
Clears memory area only when the card is unplugged from the reader (i.e. phone).

4.7. The Java Card API and Libraries

The Java Card Runtime Environment (JCRE) contains all the classes and packages that you need to create a Java Card based Applet. It can be thought of to provide the "kernel API" and this consists of three core libraries and one extension library [14]. To develop applications for a SIM card you will also often need to use some of the libraries provided by GlobalPlatform [16].

4.7.1. Package Java.lang (Java Card core library)

This library is similar to the standard Java's "Java.lang" namespace, but with many features cut away. For example, the "object" base class only contains the "equals" method and a standard constructor. The "*Throwable*" class is the base class for all exceptions; however exceptions CANNOT carry any properties as in normal Java. This makes error handling quite awkward.

Exception class types include *RuntimeException, ClassCastException, NullPointerException, NegativeArraySizeException, IndexOutOfBoundsException* and others. Note that there is no *String* class as Java strings cannot function without a garbage collector; you are left with managing character (byte) arrays.

4.7.2. Package Javacard.framework

Important aspects:
- Class ***Applet,*** this is the base class that all Applets have to derive from. Remember that its constructor is only executed ***once***, during the applications installation time.
- Contains the ***APDU*** class that helps assembling and disassembling APDU messages and transparent TPDU protocol support for protocols T=0 and T=1.
- Class ISO7816 with TPDU constants and definitions.

- Kernel and system access using Javacard.framework.JCSystem namespace (this is the Java Card equivalent to Java.lang.system).
- Support for authentication using PIN number.

4.7.3. Package Javacard.security (base cryptographic library)

This contains the base classes for Java Card encryption and public key infrastructure support which works in a similar fashion to their standard Java counterparts. A deeper discussion of this subject is outside the scope of the thesis, additional information can for example be found at [44].

Important aspects:
- Encryption key factory class *KeyBuilder*.
- Symmetric encryption key support, DES is mandatory (AES exists in newer implementations).
- Asymmetric key support using DSA/RSA.
- Random data generation (secure random).
- Hash generation.
- Signature generation and verification.

4.7.4. Package Javacardx.crypto (extension cryptographic library)

Many stronger cryptographic methods are restricted by US export rules and are thus placed in this library, whose content can vary between country (and vendor).

4.7.5. Package Javacard.rmi

This is an inter-application communications package and interface declaration for methods that can be invoked through the Card Acceptance Device (CAD). For a SIM the CAD is typically the phone itself but it can also be a smart card reader or similar. This package was first supported in Java Card 2.2.1.

4.7.6. GlobalPlatform libraries

These packages exist in UICC-based cards only. For (U) SIM cards they are maintained by the 3GPP organization. See the 3GPP TS 31.130 specification and ETSI TS 102 241 for more information.

4.7.7. Package sim.toolkit

Important aspects:
- Interface ***sim.toolkit.ToolkitInterface*** (3GPP standard 3GPP TS 43.019 [30]). Implement this in your Applet; it contains all events that can be implemented in your Applet (like received SMS, menu selection done, timer expiration and many more).
- Interface ***sim.toolkit.ToolkitConstants*** (3GPP standard 3GPP 51.014 [31]); Contains the IDs for the proactive commands and their substructures (various TLVs and TAGs).
- Package ***sim.toolkit.ToolkitRegistry*** which can be used to register your root menu items and which events your application wants to receive (see ***ToolkitInterface***).

- Package ***sim.toolkit.access*** (3GPP standard 3GPP TS 51.011 [32]) for smart card file system access and GSM functionality.
- Package ***sim.toolkit*** (3GPP standard 3GPP TS 51.014 [33]) for TLV/APDU formatting, registering events to receive from the phone and proactive command methods. TLV stands for Tag Length Value (I.e. identifier (Tag), length of data and data itself).

4.7.8. Package sim.access

This provides access to GSM data and the associated file system (see the GSM 11.11 specification).

4.7.9. Package uicc.access

Provides file system access, see specification 3GPP TS 31.130.

4.7.10. Package uicc.toolkit

This contains protocol command helpers (APDU and (BER) TLV command formatters) and functions for event registration. See ETSI TS 102.223 and Appendix B for more information.

4.7.11. Package uicc.usim.access

This contains additional constants and helpers for USIM access.

4.7.12. Host Java smartcard access packages

There are a number of different packages available to help the host (PC side) to smart card communication; some of them are freeware (search for *GlobalPlatform* on sourceforge.net).

4.7.13. Package Javacardx.smartcardio

This is a package for APDU and BER/TLV formatting and general smart-card communication [34]. It contains helper classes like ***Javax.smartcardio.CardChannel*** that is used to transmit APDU messages to and from the terminal. Another one is ***Javacard.smartcardio.CommandAPDU*** that is used to help formatting a command (C-APDU) and ***Javacard.smartcardio.ResponseAPDU*** to help formatting responses (R-APDU).

4.8. Applets and packages on the card

A package is a number of compiled classes placed in one file (like JAR files in Java). The classes might contain one or more Applet classes (making it an application) or are used as a library. An Applet class *always* has to exist inside a package file. When an Applet is installed on the smart card it is also instantiated (persistent objects and files are created when the Applet class constructor is called).

4.8.1. AID – Application Identifier

All three categories (package, Applet, Applet instance) have to be uniquely defined; this is done by giving each item an AID which is up to 16 byte long identifier, defined by the ISO 7816-5 standard. This number is split in two parts, the first 5 bytes is registered to an *application provider* (RID) which is typically the software vendor ID and the rest 11 bytes are a proprietary application identifier extension (PIX) that identifies the application for the vendor. Thus the AID is supposed to be *unique* world-wide for a specific application. The RID is normally given to you by an ISO sub-organization (see section 3.1); however a special RID value range is reserved for use with test applications.

Package AID –A package is basically the analogy to Java JAR file containing a library. Packages can refer/link to each other using the AID.
Applet AID – Each Java Applet has a unique AID.
Instance AID – Each Applet is instantiated and configured separately. More than one instance can be done using the same base Applet, thus the instances also needs a card unique AID. The generation of an instance is also often configured per SIM to include unique encryption keys and other items, whereas the Applet and Packages binaries (and AID values) are exactly the same for all deployed SIMs.

4.8.2. Pre and post issued Applets

The vendors often bundles the JCRE with their own packages and applications. These applications are often allowed to directly access the Smart Card OS using “C” and assembly language extensions since they are controlled directly by the vendor. Such applications and packages are called *pre-issued* (usually placed in ROM, such applications are called *masks*). These were basically the only type available before the GlobalPlatform standard came along.

If you are not ordering custom created cards from the vendor then you can only develop *post-issued* applications, running in the JCVM environment.

4.9. Compiling and uploading packages and Applets

4.9.1. Compiling an Applet

Your application’s source Java files are first compiled into standard Java binary class files. These class files are then converted to the limited and more compact binary format used by Java Card and bundled with various configuration files and then placed in a CAP file. The CAP files are the analog to the Java class files.

There are some things done by the CAP converter that are normally done by the JVM application loader:
- *Byte-code and class data verification* [18].
- *Allocation of static data.*
- *Compacting of symbolic references.*

- *Byte-code optimization and compression.*
- *Generate package/Applet export file (in addition to CAP file).*
- *Links in external packages using their export files. Linking on the card uses the package AID.*

Some vendors use vendor specific binary file formats, so the CAP file (and its contents) might have to be converted in yet another step.

4.9.2. Uploading an Applet to the SIM, authentication schema

The smart card's file system is typically protected from access if you do not perform an *authorization transaction* before uploading/installing applications (see Appendix C). This involves configuring a PIN code. The encryption type used is typically 3DES. After this has been done the application file and the wanted application and target type can be selected and the actual application upload can be performed.

In order to overwrite an already existing application (with the same AID) you first have to delete it, select *application delete* and enter the instance's AID, then delete the application and last the package. In the Gemalto Development Suite IDE you can see that for each step done you will see the APDU packages being sent back and forth between PC and card (see Appendix C for a list of the authentication commands). At the end of any process the card normally returns "90 00" (hexadecimal), denoting *command success*. Anything else indicates a failure of some sort, sometimes it can help by resetting the card and trying again (see Appendix B). There is no difference between uploading the Applet to a real SIM or to a SIM emulator; however the SIM emulator will not keep its internal state if you exit it.

4.9.3. Installing an Applet onto the SIM

Each application has an individual set of parameters, like the SIM-Card's root menu setup and ordering and application parameters (this acts somewhat like a "command line", typically up to 128 bytes, which can be read and used inside your application). Once the application has been uploaded and installed it needs to be reset in order for the application to be executable (the SIM emulator does this automatically). The application should now show in the phone emulator. Sometimes you also have to reset the card if your application crashes or any of the above process stages fails.

4.9.4. Applet startup, registration and execution

During phone/SIM card boot-up all Applets on the SIM card are initialized. They can either register themselves to the root menu of the SIM card or be forced to be started immediately (a master application). That SIM card application can then be started immediately when you switch on your phone, overriding even your phone's standard menu.

5 Compiling and testing the example application

Today smart card application development is typically done inside an integrated IDE where the compiler; source code editor, debugger, various simulators and application deployment are all integrated. There are a number of different frameworks/application development environments on the market like Gemalto Developer Suite, Pro-card [25] or Net Beans. For the example project the Gemalto Software Developer Suite was selected since the author has had previous experience with it.

This software suite is based on the industry standard *Eclipse* for its integrated development environment (IDE) which should be familiar to most (Java) developers today. However, the free evaluation version is restricted so you cannot download your software onto any physical SIM card. There is a software GSM phone SIM-Card simulator included (the full version also has emulators for 3G cards, CDMA and others). The evaluation version can be downloaded from their home page at *http://www.gemalto.com*.

5.1. Installing the Gemalto developer suite

First download the Gemalto developer suite from their homepage [22]. The used version during development was 3.4 (current version as of writing is 3.6, there should be no compatibility problems using it). Begin by installing the executable, when the developer suite starts you will be prompted to create a *workspace* (well known for everyone that is used with the Eclipse IDE). This is basically the directory where your projects will be placed. Please read the setup document found at

http://www.gemalto.com/products/gemxplore_developer/download/developer_suite_getting_started.pdf

for help about getting you set up (this is outside the scope of the thesis).

5.2. The example project JavaCardExampleR99

The example application provided with this thesis is called *JavaCardExampleR99*; the source code should work with any vendor. It was built for a SIM R99 card and provides some basic functions like dialing the specified number and sending an SMS, it also shows how to display texts and get input from the user (phone).

5.3. Unpacking and setting up the example

Unpack (for example using 7zip) the JavaCardExampleR99.rar file into the workspace directory you selected, the extraction creates a JavaCardExampleR99 directory (with subdirectories and files).

Importing the example project into the eclipse workspace.

Inside the Gemalto Development Suite you have to import the project into the workspace (see Figure 9).
Right click in the “Package Explorer” area and select “import”.

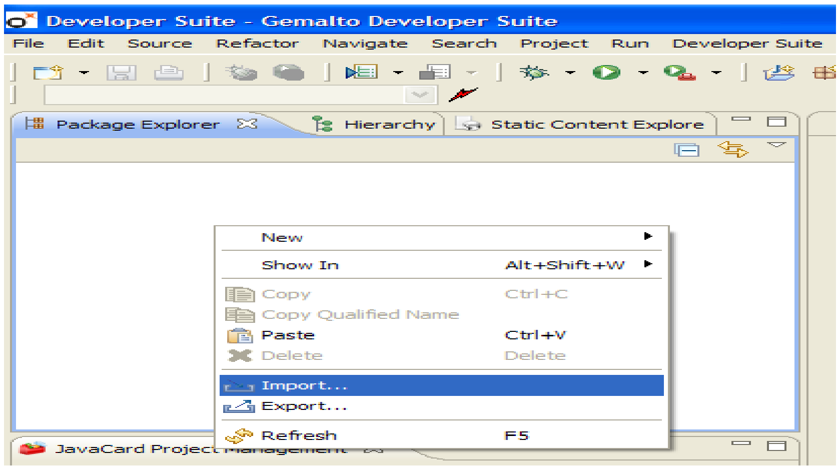


Figure 9: Importing a project into Gemalto Developer Suite

Then choose to import an existing project and press next, as seen in Figure 10.

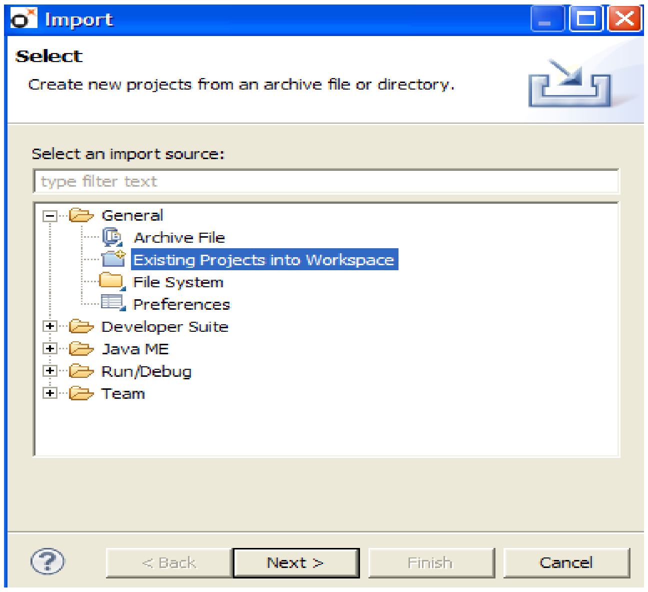


Figure 10: Importing an existing project

Press the "browse" button to the right of "select root directory", go to the JavaCardExampleR99 directory you just extracted and press OK (see Figure 11). The project will now be shown in the project list, press "Finish".

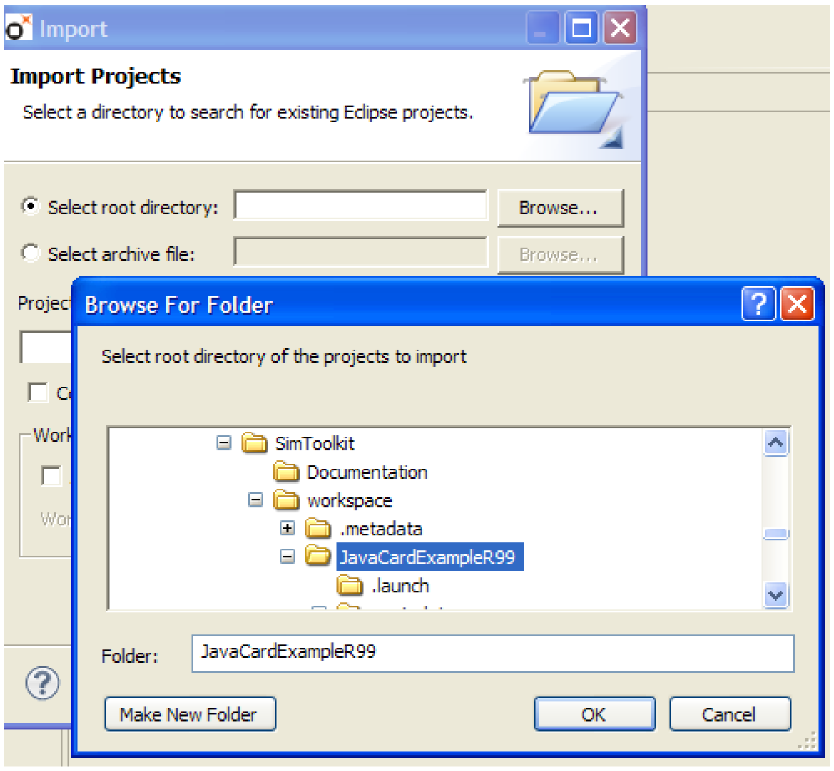


Figure 11: Selecting folder

5.4. The example project's file structure

Now the project will be shown in the package explorer (Figure 12).

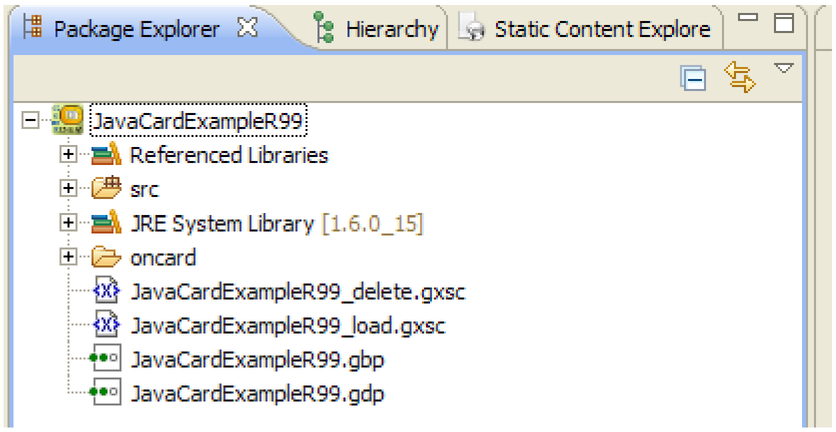
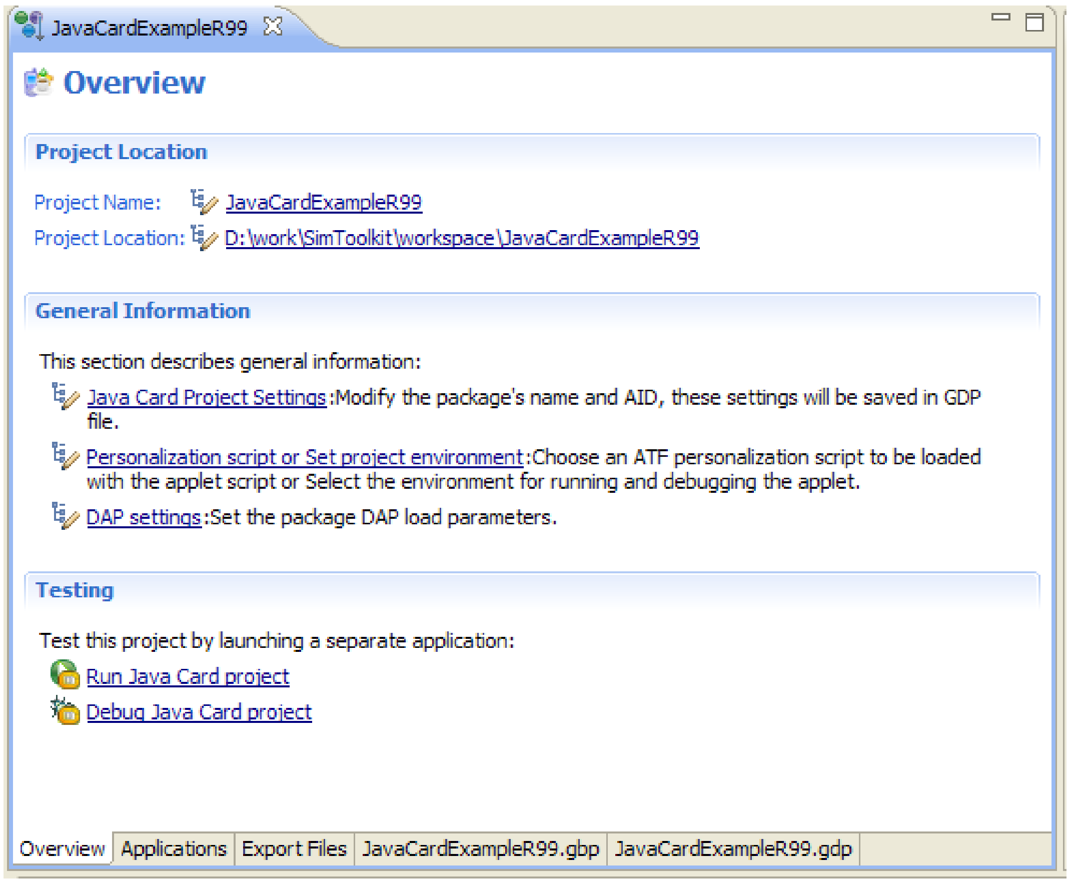


Figure 12: Package Explorer, not everything in the file system is shown here.

5.4.1. Project files (in project directory root)

File *JavaCardExampleR99.gbp* is the main build file (think make/ants). In the Eclipse environment you can change various build settings by double clicking this file inside the IDEs solution explorer (Figure 13). You can for example change the Applet's package name here or set Applet/package/instance AID (under General Information select *Java Card Properties*). These settings are actually injected into all of the *gbp*, *gdp* and *gxsc* files by the Gemalto IDE; you should not change in these files manually.


JavaCardExampleR99

Overview

Project Location

Project Name: JavaCardExampleR99
Project Location: D:\work\SimToolkit\workspace\JavaCardExampleR99

General Information

This section describes general information:
Java Card Project Settings:Modify the package's name and AID, these settings will be saved in GDP file.
Personalization script or Set project environment:Choose an ATF personalization script to be loaded with the applet script or Select the environment for running and debugging the applet.
DAP settings:Set the package DAP load parameters.

Testing

Test this project by launching a separate application:
Run Java Card project
Debug Java Card project

Figure 13: Project settings, reached by double-clicking the .gbp file.

The file *JavaCardExampleR99.gdp* contains the Applet/package/instance AID and where to find the binaries. It is used primarily by the Java class to vendor specific binary conversion tool. The files *JavaCardExampleR99_load.gxsc* and *JavaCardExampleR99_delete.gxsc* are auto-generated files that contain which APDU command sequences to use when uploading and/or deleting the applications on the card. Their content is heavily dependent on the SIM card type (R99 in this case).

The SIM card type is selected in the wizard invoked when creating a new Java Card project. Configured Applet AIDs and other settings are also injected into this file by the IDE. If you open the directory with file explorer you will see some other files and directories. Most of these are related to Eclipse project management (.classpath, .metadata etc.). For more information please refer to the Eclipse documentation.

5.4.1.1. Directory "src"

Java card source files. Under this directory there are similar directories as the package paths (in this case *jcexample*).
The example Applet's source file is ***src/jcexample/JCExample.Java.***
Note that package names should always be in lowercase.

5.4.1.2. Directory "classes"

Compiled class binaries (created by SUNs Java compiler). These are in turn used to create the resulting binaries (see "oncard" below).

5.4.1.3. Directory "oncard"

This dictionary contains the binaries that can be uploaded to the card and other compiler output.
The example projects binary files are placed under ***oncard\jcexample\Javacard.***
The EXP, JCA, SAP, CAP and JAR files[8] are created by SUN's Java class to Java Card binary converter tool. The IJC is created with the *makeijc* tool.

- ".CAP" files are converted Java binary class files (there can thus be many of them).
- ".EXP" files contains the public interfaces for the classes in the corresponding CAP, this file type is basically for build tools only, they are not uploaded to the card.
- ".JCA" files (ASCII) are for helping debugging (basically a disassembly of the CAP binary).
- ".JAR" files are libraries consisting of one or more CAP files (zipped together).
- ".IJC" files are an extra compressed/stripped form of JAR files; containing only items that are according to the SIM Alliance standard.
- ".SAP" files are used by the GSM simulator (instead of CAP file).
- ".install" files contain AID and application initialization parameters (like memory usage size).

[8] *See the Gemalto Developer Suite setup document for more information*.

5.5. Building and starting the example project

The example project is set up to be debugged with an emulated R99 SIM in a virtual GSM phone. There are other ways to test the application (with real SIM in virtual GSM phone or real SIM in a real phone etc., please refer to section 5.9).

The application is built in several steps, first the code is compiled to class files and then these are converted to a number of different files (section 5.4.1.3) as seen in Figure 14.

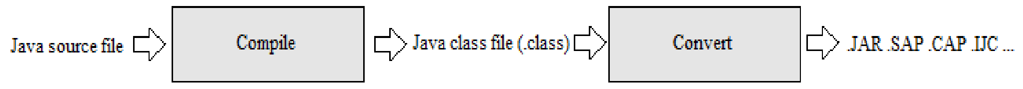
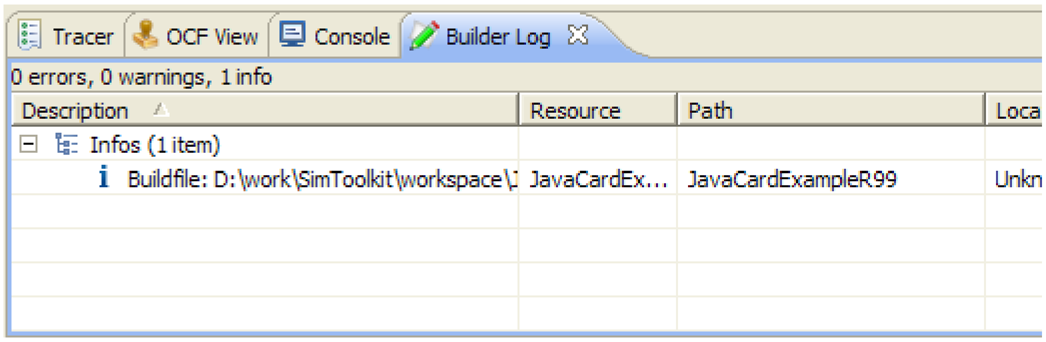


Figure 14: The build process.

Once the build is done the binaries can be uploaded onto the card or into the SIM emulator. The SAP file is used by the emulator and the CAP or IJC file for uploading to the actual smart card. After uploading the application can be personalized (configured) and then installed (constructor executed).

Press *F11* or *Run / Debug* in the main IDE menu, and then select *Java Card Project*. The project will build automatically, if successful the deployment files are created and the simulator spawned.

You will see a number of messages appearing in the Console / OCF view. Any errors will be shown in the "Builder Log" view, see Figure 15 below.

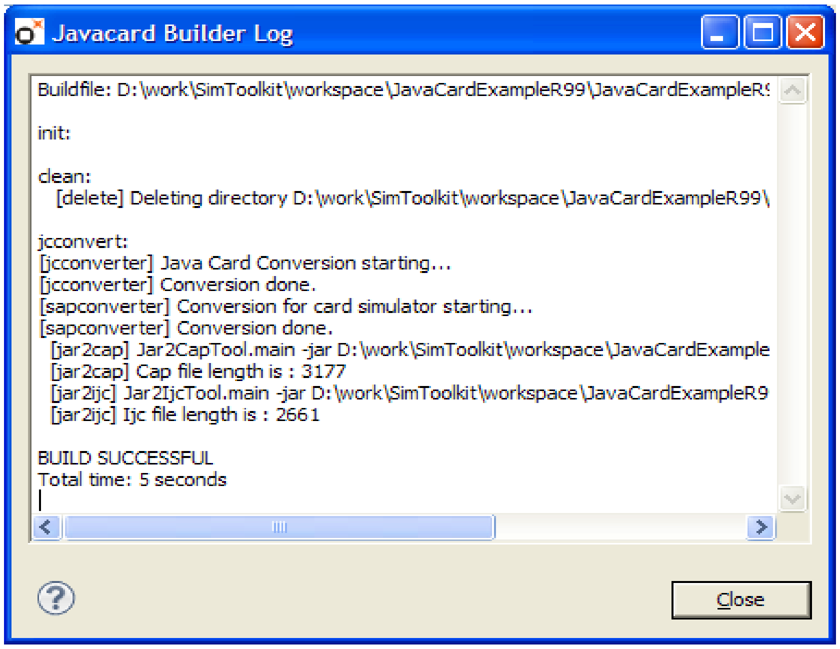


0 errors, 0 warnings, 1 info

Description	Resource	Path	Loca
Infos (1 item)			
Buildfile: D:\work\SimToolkit\workspace\J	JavaCardEx...	JavaCardExampleR99	Unkn

Figure 15: The picture shows the result of successful build in the lower right pane.

If there are any build errors there will be a little exclamation mark beside the project shown in Package Explorer (and the Builder Log will show an error icon). If you double-click this file the build result log will be shown as below (Figure 16).



```
Buildfile: D:\work\SimToolkit\workspace\JavaCardExampleR99\JavaCardExampleR

init:

clean:
   [delete] Deleting directory D:\work\SimToolkit\workspace\JavaCardExampleR99\

jcconvert:
[jcconverter] Java Card Conversion starting...
[jcconverter] Conversion done.
[sapconverter] Conversion for card simulator starting...
[sapconverter] Conversion done.
  [jar2cap] Jar2CapTool.main -jar D:\work\SimToolkit\workspace\JavaCardExample
  [jar2cap] Cap file length is : 3177
  [jar2ijc] Jar2IjcTool.main -jar D:\work\SimToolkit\workspace\JavaCardExampleR9
  [jar2ijc] Ijc file length is : 2661

BUILD SUCCESSFUL
Total time: 5 seconds
```

Figure 16: example build log of a successful build.

You will get an idea about what the error is about, however many of the details are hidden in the conversion between Java class file to Java Card CAP files (jcconverter). It is very easy to fail this step, like if you happen to use string objects or wrong type of constant arrays. Also many of the integrity checks [18] normally done by a JVM when it loads the classes is instead done by this file conversion tool, thus you can get very many kinds of error messages from it. Note that the reported line number is not always matching the offending source code line since the tool is parsing the binary class file, not the Java source code file. Errors will look something like this:

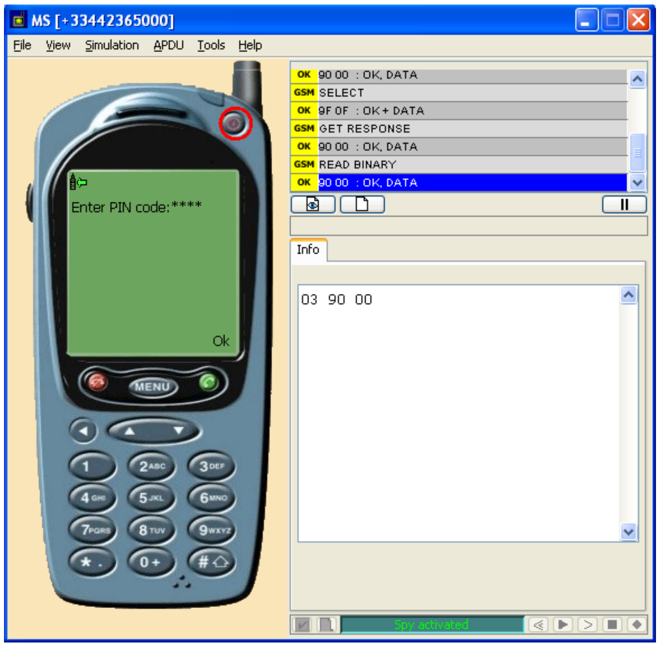
"*[jcconverter] error: line 4: examplepackage.LocationExample: unsupported parameter type String of invoked method <init>(Java.lang.String) of class Java.lang.Error.*"

Standard Java compilation errors will show directly in the Eclipse GUI as red lines in the scrollbar to the right of the source code window (your Java code is analyzed in real time by the Eclipse framework). If the application builds successfully the virtual GSM phone window will show up. The R99 SIM emulator's window is hidden in the background by default.

If you want to re-run a debugging session you typically have to kill the "Javaw.exe" processes for the R99 SIM emulator with task manager. These processes do not exit by themselves just because the virtual GSM phone application is stopped. If you do not do this there will be more and more Javaw.exe processes left running.

5.5.1. The GSM Simulator running the example project

The GSM simulator will start with a blank "screen", you have to turn on the virtual phone by clicking button marked with a red circle below, it will then ask you for a PIN as shown, enter 1234 which is the default SIM password (validated by the SIM emulator), see Figure 17.



```
OK  90 00  : OK, DATA
GSM SELECT
OK  9F 0F  : OK+ DATA
GSM GET RESPONSE
OK  90 00  : OK, DATA
GSM READ BINARY
OK  90 00  : OK, DATA
```

```
03 90 00
```

Enter PIN code:****

Ok

Figure 17: The GSM emulator booting up.

Note the APDU commands and APDU responses (in hexadecimal) in the window to the upper and lower right. You can click on a particular command on the top to see its data contents on the bottom

5.5.2. The virtual phone's root menu

The GSM virtual phone normally just has one function, and that is the option *"dial number"*. If the SIM emulator (or SIM in a smart card reader connected to the GSM virtual phone) has installed application(s) on it a "STK application" selection will also be available. This opens the SIM card root menu. The sample Applet is called "JCExample", once started you can select between three options, see Figure 18 below.

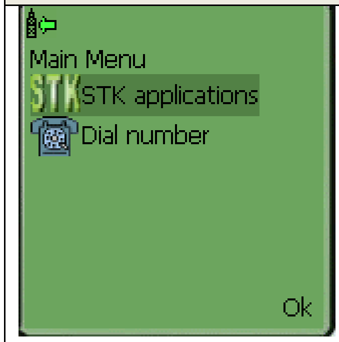
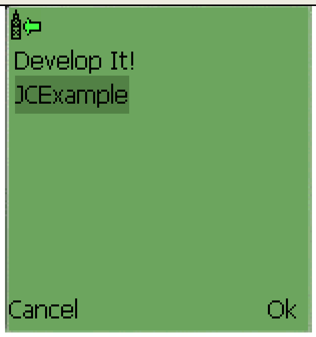
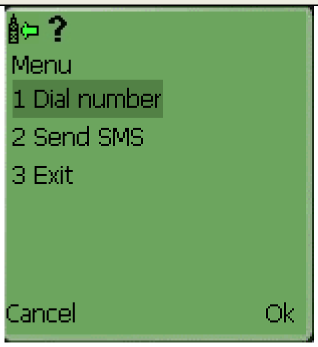
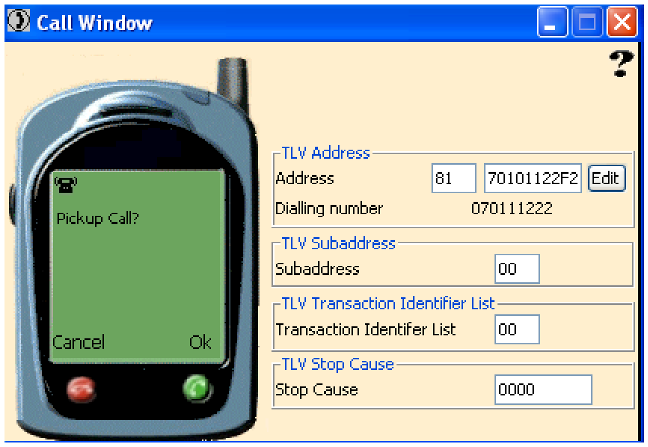
GSM Virtual phone main menu	The STK Applications menu	JCExample Applet main menu
 Main Menu STK applications Dial number Ok	 Develop It! JCExample Cancel Ok	 Menu 1 Dial number 2 Send SMS 3 Exit Cancel Ok
Select STK Application (just press enter)	Select JCExample. *NOTE: The STK Application main menu is called "Develop it!" by default, this text is stored in a GSM file system parameter.*	JCExample Applet running. Now select "Dial number" and enter a phone number.

Figure 18: Starting and executing the example application (JCExample Applet).

5.5.3. Screen shot of the virtual GSM phone "dialing" a number

If you select *"Dial number"* you are asked to enter a phone number. Once done the phone emulator shows you the number you entered to the right and other information (TON/NPI data). You can accept the "call" by pressing Enter (or the green "pickup" button), see Figure 19.



TLV Address
Address 81 70101122F2 Edit
Dialling number 070111222

TLV Subaddress
Subaddress 00

TLV Transaction Identifier List
Transaction Identifer List 00

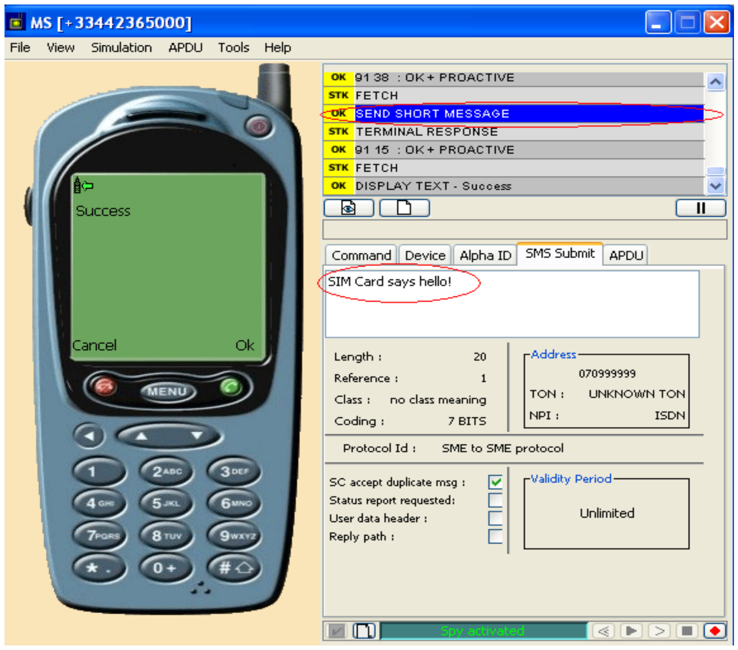
TLV Stop Cause
Stop Cause 0000

Figure 19: When dialing a number the emulator will let you select an action (pickup or reject).

The application will now say *"call in progress";* you can terminate it by pressing the red "hang-up" button. The Applets will then display "success" and go back to its main menu.

5.5.4. Screen shot of the virtual GSM phone sending a SMS

If you instead select "Send SMS" in the main menu (use the arrow keys) you are asked to enter a phone number to which a SMS will be sent. The Applet will just say "Success" (Figure 20). You can see the sent message content (*"SIM Card says hello!"*) by clicking on the "SEND SHORT MESSAGE" row to the right and select "SMS Submit" tab. Of interest here is also the full APDU command (tab to the right).



```
OK   91 38  : OK + PROACTIVE
STK  FETCH
OK   SEND SHORT MESSAGE
STK  TERMINAL RESPONSE
OK   91 15  : OK + PROACTIVE
STK  FETCH
OK   DISPLAY TEXT - Success
```

SIM Card says hello!

Length : 20
Reference : 1
Class : no class meaning
Coding : 7 BITS

Address
070999999
TON : UNKNOWN TON
NPI : ISDN

Protocol Id : SME to SME protocol

SC accept duplicate msg : ✓
Status report requested: ☐
User data header : ☐
Reply path : ☐

Validity Period
Unlimited

Figure 20: The emulator can display the contents of the sent SMS.

5.6. Example application source code overview

You should keep the source code (JCExample.Java) beside you when reading the following sections. The examples source code has extensive code comments; however some explanation is in order as to which methods that needs to be implemented and as to why. To understand the APDU/proactive commands used you should also download the GSM 11.14 specification and the ISO 7816-4 standards (available at ETSI homepage [2]). An introduction is given in Appendix D.

The Applet package

```
package jcexample; // Applet's package name
```

The Applet package name must be in lowercase and must correspond with what is set up in the GDP file; otherwise the Applet will not load.

The used packages

```
import sim.toolkit.*;
import Javacard.framework.*;
```

After the package definition the used packages are included, just like in any Java application.

5.6.1. The Applet main class

The main core of an Applet application is a class inherited from the framework class ***Javacard.framework.Applet***. This class contains the initialization, setup, shutdown and event handling methods of your application. The event handlers are called when the JCRE receives APDU commands and are declared by the ***sim.toolkit.ToolkitInterface*** interface. Another interface that comes in handy is sim.toolkit.ToolkitConstants which contains various constants used by the toolkit and APDU messages.

```
public class JCExample
  extends Javacard.framework.Applet
  implements ToolkitInterface, ToolkitConstants
{
}
```

5.6.2. Constants used in an Applet

Constants should always be declared static final to avoid ending up in the persistent memory area.

```
private static final short MSG_MAINMENU_OFFSET = (short)
0;
```

Text constants must be placed in persistent memory however. Typically these texts are in 8 bit ASCII and not UCS2 (16 bit) to save memory space. The byte casts are unfortunately needed here.

```
private byte[] messageStrings =
{
```

```
(byte)'J',(byte)'C',(byte)'E',(byte)'x',(byte)'a',(byte)'m
',(byte)'p',(byte)'l',(byte)'e'
}
```

5.6.3. The class constructor

```
public JCExample()
```

The constructor of your Applet will be executed only ONCE during your applications lifetime, namely during the installation phase. Here the various persistent data is allocated (ONCE) and menu items are added to the Applet's root menu (done through the class ***ToolkitRegistry***, function initMenuEntry).
The Applet also registers which events it wants to listen/subscribe to here.

5.6.4. The static *install* method

```
public static void install(byte bArray[], short bOffset,
byte bLength)
```

This method must exist in your Applet main class. It is called by the framework when your Applet is installed (after it has been downloaded onto the card). The arguments parameter contains the Applet's "command line", a series of initialization bytes for this instance of the Applet declared when you configure the instance for download. The install method should create ("new") an instance of your Applet (which obviously invokes the constructor, see above) and then call the *register* method on it which registers your Applet instance with the card.

5.6.5. The *processToolkit* method

```
public void processToolkit(byte event)
```

This method gets called by the Java Card operating system when an event happens. Some events always happen when your application is selected but most of them you have to subscribe to explicitly. The "EVENT_PROFILE_DOWNLOAD" event happens when the application has been initialized the first time, it is a good and safe place to check for necessary phone/SIM capabilities.
The "EVENT_MENU_SELECTION" is trigged when the user selects a root menu entry that corresponds to your application.

5.6.6. The *dialNumber* method

This method is provided as an example how to construct a C-APDU using a simpler BER-TLV structure (multiple small TLV sub-structures, see Appendix B). The command is sent with the ProactiveHandler's "send" method.

5.6.7. The *sendSMS* method

The *sendSMS* method gives good insight into more complex APDU command composition. It is further described as a formatting example at the end of Appendix B.

5.6.8. Other methods

See the source code. There are examples of how to display text on the phone, get user input and sending a SMS and some simple event handling. They are properly documented with code comments.

5.7. How to test your SIM application

5.7.1. Using a SIM emulator

The Gemalto Developer Suite provides various SIM emulators (for respective (U) SIM family such as R99, R5 and R6). In fact the evaluation version does not allow you to download your application to a real SIM card. The emulator is a stand-alone Java application that starts in its own window. It even lets you debug your application and provides all the usual stuff you would find in a Java software debugger.

One thing to note about the SIM simulators is they are *MUCH* more forgiving and flexible than typical smart card hardware. All features are supported and it behaves in a more predicable way than real hardware typically does. So beware, even if your application works fine in a simulator it might not work at ALL on a real SIM. Typical cause of this is due to memory allocation and access restriction/problems.

5.7.2. Using a physical SIM Card

To connect a SIM card to your PC you need a smart card reader (also called CAD, “Card Acceptance Device”). It connects with RS232 or USB. Quite possibly you have to install drivers for it that came with the package. Microsoft Windows contains a smart card API that the Gemalto Suite uses.

You use the JCardManager tool found under the “Developer Suite” option in the main menu, you are prompted to select the SIM card reader to use. Select the correct SIM card type in the menu. Finally select “op201p” and upload the application (easiest with the Quick Load option).

Note that this option only works with the full version of Gemalto Developer Suite, not the evaluation version. The configuration of the SIM parameters is outside the scope of this article; please refer to the documentation that comes with Gemalto Developer Suite (Help Option).

5.7.2.1. Using an imprinted SIM with the virtual phone

It is quite cumbersome to upload software into a real SIM card, unplugging it from the SIM smart card reader, inserting it into a real phone, switching it on, finding the application in the phone’s menu and finally starting it. Because of this the Gemalto suite contains various phone emulators (GSM (2G)/3G, CDMA and so on). When started they shows a GUI of an imaginary phone, with keyboard, display, SMS functionality and so on.

It also provides access to APDU logs and other developer utilities that a real phone does not provide. The phone emulator can be configured to either use a simulated (“virtual”) SIM or to use a *real* SIM card connected to your pc over USB/serial cable via a smart card reader.

You then “turn on the emulated phone by pressing the power-on icon (see section 4.2). Your SIM application registers itself and shows up in the menu tree during boot-up. You should now be able to select and start it from the phone’s starting menu. If there are errors during installation or when executing applications initialization code it typically does not show at all.

5.7.2.2. Using an imprinted SIM in a physical phone

Please note the restrictions as outlined in section 3.4.1, not all APDU commands works on all handsets which might render your Applet nonfunctional. On older handsets the standards implementation can also be quite buggy. If you do a commercial quality Applet you will end up having to test it on a whole bunch of handsets from many different vendors.

When you place the SIM in a real phone you can reach the SIM cards root menu through various methods, but this is very phone vendor and version specific. For example on older Ericsson phones the root menu appears under “Games”, for some Nokia phones a new selection appears directly in the phone’s main (root) menu. There seem to be no standard governing this; you have to investigate by yourself. The SIM root menu will typically be called *“Develop it!”* by default if you use Gemalto development cards. The SIM card is quite robust and won’t break by being plugged in and out of the SIM card reader. Most phones needs to be powered off before plugging in a SIM but if the phone allows it a “hot swap” is possible.

6 Results and analysis

This section explains the results and experience of the development process during implementation of the attached sample application. It also sums up the experience the author have had developing other commercial SIM applications to give this discussion a broader picture.

The example application is just meant as an introduction to the complex field of smart card and SIM card application development and thus kept very simple, its basic functions are:

- Display a menu on the handset and allow user to select an option.
- Display a text on the handset.
- Ask user for input (here a phone number).
- Dialing the specified number
- Sending of a binary SMS.

More advanced functions like encryption, signature checking, PKI, TCP/IP communication and event monitoring was not implemented. To make any sense of such functionality a server-side (i.e. PC) application would also need to be developed. This was not done, both to keep the scope down to a manageable level and to limit the length of the thesis. Nevertheless the example application’s source code should give you some insight about how the APDU commands work and how to construct them from its elements (TLV and BER-TLV composition).

The end result CAP file is around 3k in size and the more compact IJC version is 2.6k which fits into any R99 compatible SIM card. The simple example application’s comparatively big size should alert you to the fact that it can be quite hard to fit any *real* Applet (with business logic, transactions and so on) onto the card. It is even worse if you have to share the constricted space (32k or less) with other applications.

Primary problems encountered when implementing the application:

- The BCD packing of phone numbers for dialing and sending SMS was tricky. The author had some problems making the packing and setup of TON/NPI work correctly. There are helper classes for this; a real Applet should use them.
- The composite structure used sending for binary SMS took some time to get right (mainly because of the padding requirements). Any error in the request structure just results in an error result code back which was not very helpful.

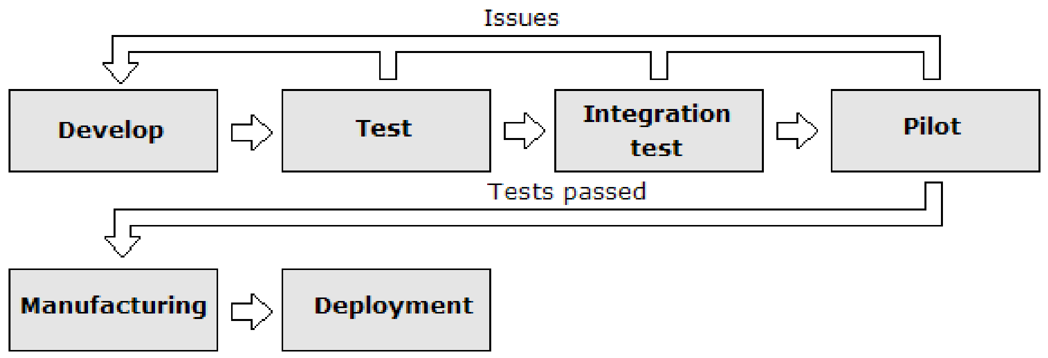


Figure 21: Typical development process

Figure 21 shows the typical development process for a commercial grade application. The next sections will discuss this step by step in the context of SIM application development. Even a commercial project involving a seemingly simple application can take substantial time and effort to complete.

6.1. The application development phase

The authors experience is that most of the coding effort goes into creating various helper wrappers for the APDU commands (packing and unpacking APDU data structures and making sure these really works in the anticipated way). Once you get a library of methods ready you copy and paste these and then modify them for this applications particular need (if you are lucky you might have enough space to add a library package as a ROM mask, freeing up valuable Flash/EEPROM space).

Modification of the copy-pasted code is typically necessary since most methods will need to share one common transient (RAM) work buffer. Working this way is both tedious and error prone. At every point you have to keep track of which method that uses what buffer areas so they don’t overwrite each other’s data, often forcing you to incorporate some kind of stack like scheme.

Keep the compiled binary’s size small. This mean *all* non-used code (debug methods and so on) has to be removed at the end of the development phase. More than once you will feel your final optimized Applet code looks more like assembler than Java since most of the OOP concept has been lost. Hopefully this situation will change once Java Card 3 based cards becomes the de-facto standard (the author has no experience with Java Card 3 but on paper it looks very promising).

6.2. The application testing phase

When an error is found during testing the question is often raised if it was the developer that made a mistake or if it was caused by a bug on the smart card or some nonstandard behavior on the handset. SIM card emulators are usually working both faster and better and are more forgiving to your mistakes than a real card ever would be. If you got your Applet to work in the emulator then you can be almost sure it will *not* directly work on a real smart card.

One common family of bugs manifest like the Applet refuses to appear in the root menu. These are usually caused by failure in the constructor/registration logic and are almost always due to memory allocation issues. Be especially aware about memory constraints if you know other Applets might be installed together with yours on the card (the Card Issuer is usually handing out memory quotas to control this).

Another family of bugs is caused by problems with APDU formatting. These make your Applet to hang or crash unexpectedly. For safety reasons the application is then often locked out, forcing you to delete it and reinstall. You can debug an application with single-step, breakpoints and so on in an emulator but when running on a real smart card it is considerably harder. Often you have to fall back on the classical *"printf"*-strategy (infamously known from "C") and do repeated install-test-delete cycles. This takes time and after some repeats causes severe frustration.

For a commercial grade application it is very important that you have full unit tests in place and simulations that will follow *every* flow path in the application. This is not only to test the logic per-se but also so you do not take too much stack space and that the shared work buffer usage works for all executable combinations. Often applications are of a sensitive nature so hackers will try to perform *every possible* combination of user input there is in order to exploit any weaknesses. To be safe you more or less have to mathematically prove that your code is sound. If your application is to be pre-installed (ROM mask) then you cannot fix any issues once the application is deployed. This is a bit like constructing software for a NASA probe that will be sent away to some distant planet.

6.3. The integration-test phase

Here you have to test your application on all combinations of SIM cards (vendors and so on) that you will have to deploy to. As such these can have different available memory sizes and performance characteristics; sometimes they come with a list of known bugs (errata) that you need to be aware of.

The next phase is handset testing. Like previously mentioned some older handsets does not implement all APDU commands, be sure that you check the capability information. Older handsets might also implement the APDU commands in a buggy way, especially if you use some rarely used feature. To be really sure you have to test each SIM card type in each handset model which can be very hard to do.

6.4. The pilot phase

Usually the application has to run quite some time as a pilot, just enabled for some few test subscribers. In order to enter the pilot phase the operator of course have to verify that all their possible combinations of SIM card applications run as they should together with yours and that your code does not contain anything possibly malicious.

6.5. The manufacturing phase

Once your application has passed all previous steps it is time to manufacture the SIM cards (section 2.4). Of course this step is not needed if you intend to post-issue it or deploy the application over the air (section 2.5). The Card Issuer usually performs the post-issuing phase and can then use an off-the-shelf SIM card. Pre-issued applications (placed in ROM) are much more complex and needs to involve the manufacturer.

6.6. The deployment phase

Either the new SIM cards are handed out physically or the Applet is deployed OTA. The latter also gives the opportunity to make an upgrade if any bugs are found.

7 Conclusion

The background work needed for writing this thesis has been extensive and taken much more time than the author originally thought it would need. Often the obtained information has been shown to be incomplete, incorrect or outdated. The various standards have been shown to be quite interwoven and branch out in all directions which are to be expected based on their iterative evolution. The standardization process is consensus driven and might not always be based on the best technical solution since it is primarily driven by commercial interests.

The author believes the primary goal of the thesis has been reached, namely to provide both a good theoretical and practical introduction to the field. The emphasis has been on the theoretical part and it has been given much more space. However, this was done by purpose in order to give a proper foundation for future studies.

7.1. Method criticism

Extensive usage of web resources has worked very well and links are provided in the references section for further and deepened studies. Using the Internet as a primary resource is always a risky business since sources might not have been verified and is thus more error prone than published work. This means that errors *might* have followed into this text. The author has tried to locate and make use of published academic work where possible.

7.2. Future studies

The references section contains lots of good links and articles that should be read if you want to deepen your knowledge further.

Documentation and discussion groups can be found at Oracle's Java Card pages and at Gemalto's developer pages. Gemalto also hold courses in SIM application programming. The Java Card Forum's homepage at *http://www.Javacardforum.org* is an excellent resource of information and source code.

A good but a bit aged book (published 2001) about SIM application programming is "A*pplication development with SMS and the SIM Toolkit"* by S. Guthery and M.Cronin (ISBN 0071375406 / 9780071375405).

Further Java Card 3 information can be found on Oracle's homepage. Michel Koenig gives an excellent overview of its features plus gives examples at his homepage [58].

Reference list

[1] ISO (International Organization for Standardization) homepage (ISO standards)
http://www.iso.org
(Lookup 2011)

[2] ETSI organization's homepage (ETSI specifications)
http://www.etsi.org
(Lookup 2011)

[3] 3GPP homepage (3GPP specifications)
http://www.3gpp.org/
http://www.3gpp.org/ftp/specs/html-info/1111.htm *(GSM 11.11 specification)*
http://www.3gpp.org/ftp/specs/html-info/1114.htm *(GSM 11.14, SIM toolkit)*
(Lookup 2011)

[4] IEC organization homepage
http://www.iec.org/
(Lookup 2011)

[5] GSM organization homepage
http://www.gsm.org/
http://www.gsmworld.com
(Lookup 2011)

[6] Wikipedia, *"GSM"*, *"GSM Association"* (overview)
http://en.wikipedia.org/wiki/GSM
http://en.wikipedia.org/wiki/GSMA
(Lookup 2011)

[7] Wikipedia, 3GPP2 group
http://en.wikipedia.org/wiki/3GPP2
(Lookup 2011)

[8] Wikipedia, *"American National Standards Institute"*
http://en.wikipedia.org/wiki/American_National_Standards_Institute
(Lookup 2011)

[9] ANSI (American National Standards Institute) homepage
http://www.ansi.org/
(Lookup 2011)

[10] Gsm-security.net, "*How do Authentication and Key generation work in a GSM network?*"
http://www.gsm-security.net/faq/gsm-authentication-key-generation.shtml
(Lookup 2011)

[11] Gemalto, *"The role of the UICC (sim card) in Long Term Evolution (LTE)"*
http://www.gemalto.com/uicc_role/
(Lookup 2011)

[12] Wikipedia, *"Universal Mobile Telecommunications System"*
http://en.wikipedia.org/wiki/Universal_Mobile_Telecommunications_System
(Lookup 2011)

[13] D. Deville, A. Galland, G. Grimaud, S. Jean, "*Smart Card Operating Systems: Past, Present and Future"*, NORDU/USENIX Conference.
http://www2.lifl.fr/~grimaud/Research/SelectedPaper
(Published 2003)

[14] Oracle Corporation (Sun Microsystems) *"Java Card Technology".*
Java Card specifications (also see [16])
http://www.oracle.com/technetwork/Java/Javacard/overview/index.html
http://Java.sun.com/products/Javacard/specs.html
http://Java.sun.com/Javacard/3.0/specs.jsp
(Lookup 2011)

[15] Wikipedia, *"Java Card Open Platform", "Java Card", "GlobalPlatform"*
(overviews).
http://en.wikipedia.org/wiki/Java_Card_OpenPlatform
http://en.wikipedia.org/wiki/Java_Card
http://en.wikipedia.org/wiki/GlobalPlatform
(Lookup 2011)

[16] GlobalPlatform consortium homepage (UICC/Java Card specifications and whitepapers).
http://www.globalplatform.org
(Lookup 2011)

[17] IBM International Business Machines Corporation, *"JCOP – The IBM GlobalPlatform JavaCard implementation"*
IBM Zürich Research Laboratory
ftp://ftp.software.ibm.com/software/pervasive/info/JCOP_Family.pdf
(Lookup 2011)

[18] H. Konstantin, *"Use of Cryptographic Codes for Bytecode Verification in Smart Card Environment".*
UNIVERSITY OF KUOPIO, Faculty of Business and Information Technology, Computer Science.
http://www.cs.uku.fi/~khiouppe/masters.pdf
(Published 2003)

[19] J. Iguchi-Cartigny, J. Lanet, *"Developing a Trojan Applet in a smart card".*
Journal in Computer Virology archive Volume 6 Issue 4, November 2010.
http://portal.acm.org/citation.cfm?id=1873380
(Published 2010)

[20] G. Bizzotto, G. Grimaud, *"Practical Java Card byte-code compression".*
LIFL, Universite de Lille, Gemplus Research Lab. *In Proceedings of RENPAR14 / ASF / SYMPA*
http://academic.research.microsoft.com/Publication/30591/practical-Java-card-bytecode-compression-1
(Published 2002)

[21] Infineon homepage (smart card manufacturer)
http://www.infineon.com
(Lookup 2011)

[22] Gemalto N.V homepage (smart card manufacturer).
Download the "*Gemalto Developer Suite"* from here. Site also contains lots of smart card development documentation and examples.
http://www.gemalto.com
(Lookup 2011)

[23] Visa Incorporated
http://www.visa.com
(Lookup 2011)

[24] Multos Limited homepage (MULTOS virtual machine). Contains development info.
http://www.multos.com/
(Lookup 2011)

[25] ProCard homepage
http://www.procard.pl
(Lookup 2011)

[26] TTFN.NET, "*ISO/IEC 7816 Part 4: Inter industry command for interchange"*
(APDU command format details)
http://www.ttfn.net/techno/smartcards/iso7816_4.html
(Lookup 2011)

[27] Wikipedia, *"ISO/IEC 7816"*
http://en.wikipedia.org/wiki/ISO/IEC_7816
(Lookup 2011)

[28] Cardwerk, *"The ISO 7816 Smart Card Standard: Overview"*
http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx
(Lookup 2011)

[29] D. Everett *“Smart Card Technology: Introduction To Smart Cards - Page 3”*
Smart Card News Ltd http://kbts.neic.nsk.su/satxpress/SmartCard/ISO7816-3.htm
(T0 protocol)
(1997)

[30] 3GPP, ”TS 43.019”
http://www.3gpp.org/ftp/Specs/latest/Rel-5/43_series/43019-560.zip
(Lookup 2011)

[31] 3GPP, ”TS 51.014”
http://www.3gpp.org/ftp/Specs/latest/Rel-4/51_series/51014-450.zip
(Lookup 2011)

[32] 3GPP TS 51.011
http://www.3gpp.org/ftp/Specs/latest/Rel-4/51_series/51011-4f0.zip
(Lookup 2011)

[33] 3GPP TS 51.014
http://www.3gpp.org/ftp/Specs/latest/Rel-4/51_series/51014-450.zip
(Lookup 2011)

[34] W. Mostowski, *“Global Platform for Java SmartCardIO”*
http://sourceforge.net/projects/gpj/
(Lookup 2011)

[35] Wikipedia, *“A5”* (the GSM/UMTS A5 encryption standard)
http://en.wikipedia.org/wiki/A5/1 (GSM)
http://en.wikipedia.org/wiki/A5/2 (GSM, Obsolete)
http://en.wikipedia.org/wiki/A5/3 (UMTS/3G encryption strength)
(Lookup 2011)

[36] Wikipedia *“GSM 03.38”* (additional information about GSM character sets).
http://en.wikipedia.org/wiki/GSM_03.38
(Lookup 2011)

[37] ETSI, *“ETSI and GlobalPlatform strengthen relationship”*
Over the Air concepts
http://www.etsi.org/WebSite/NewsandEvents/200909_GlobalPlatform.as
px
(2009)

[38] Wikipedia, *“Alliance for Telecommunications Industry Solutions”*
http://en.wikipedia.org/wiki/Alliance_for_Telecommunications_Industry_Solution
s
(Lookup 2011)

[39] Wikipedia, *“3GPP”* (3G and 4G/LTE historical evolution)

http://en.wikipedia.org/wiki/3gpp
(Lookup 2011)

[40] Wikipedia, *"3GPP Long Term Evolution"* (see also[3], [39])
http://en.wikipedia.org/wiki/3GPP_Long_Term_Evolution
(Lookup 2011)

[41] Wikipedia, *"Short message service center" (SMSC)*
http://en.wikipedia.org/wiki/Short_Message_Service_Center
(Lookup 2011)

[42] Wikipedia, *"Subscriber Identity Module" (ICC-ID)*
http://en.wikipedia.org/wiki/ICC-ID
(Lookup 2011)

[43] Seamless Distribution AB homepage (*Products: Mollet, GoHandset, GoShopping, GoBanking etc.*).
http://www.seamless.se
(Lookup 2011)

[44] T. Elo, *"A Software Implementation of ECDSA on a Java Smart Card"*
Department of Computer Science, Telecommunications Software and Multimedia Laboratory, HELSINKI UNIVERSITY OF TECHNOLOGY.
http://amadousarr.free.fr/crypto/ECDSAJAVACARD.pdf
(Published 2000)

[45] M. Erlandsson *"Smartcard based heart-beat service for M2M communication"*
Department of Computer and Information Science, Linköpings Universitet.
http://www.uppsatser.se/uppsats/c8b18501aa/
(Published 2010)

[46] BSNL (Bharat Sanchar Nigam Ltd), *"Value Added Service"* (example of operator Value Added Services).
http://www.bsnl.co.in/service/mobile_vas.htm
(Lookup 2011)

[47] Vasco.com homepage, "*Digipass nano"* (e-signatures).
http://www.vasco.com/products/digipass/digipass_software/digipass_nano.aspx
(Lookup 2011)

[48] Invigo.com, *"Call completion"* (and other VAS).
http://www.invigo.com/solutions_call_completion.php
(Lookup 2011)

[49] Vodaphone, *"SmarTone-Vodafone IDD & Roaming SIM card "* (VAS).
http://www.smartone-vodafone.com/jsp/mobile/prices/store_valued_SIM/english/smartone_idd.jsp

(Lookup 2011)

[50] J. Roujinsky, *“Automatic Call Completion”* (patent application)
Reinhold Cohn and partners.
http://www.sumobrain.com/patents/wipo/Method-system-automatic-call-completion/WO2008050325.html
(Published 2007)

[51] Simage technologies, *“Roaming management”*
http://www.simagetechnologies.com/Content/Roaming-Management-23.html
(Lookup 2011)

[52] N. Jha, *”Death of the SIM card near” (*SoftSim standard in progress)
Themobileindian.com
http://www.themobileindian.com/news/611_Death-of-the-SIM-card-near
(Published 2011)

[53] *NFC Technology overview*

Wikipedia, *“Near Field Communication”*
http://en.wikipedia.org/wiki/Near_Field_Communication
(Lookup 2011)

N. Warkagoda, *“Near Field Communication (NFC), Opportunities & Standards”*
Telenor
http://www.umts.no/files/081028%20nfc_standards_payments%20Narada.pdf
(2008)

[54] p. Kumar, *“Can NFC make mobile wallet a reality?”*
Themobileindian.com
http://www.themobileindian.com/news/693_Can-NFC-make-mobile-wallet-a-reality
http://en.wikipedia.org/wiki/NFC
(Published 2011)

K. Heussner, *“Is Your Next Credit Card Your Cell Phone?”*
http://abcnews.go.com/Technology/credit-card-cell-phone/story?id=12757937
abc NEWS
(Published 2011)

[55] Wikipedia, *“Radio-frequency identification”*
http://en.wikipedia.org/wiki/RFID
(Lookup 2011)

[56] *Articles discussing RFID integrity issues*

N. Lomas, *“RFID could be in all cell phones by 2010”*
Zdnet.com
http://www.zdnet.com/news/rfid-could-be-in-all-cell-phones-by-2010/315292

(Published 2009)

JG Mason, *” No escaping RFID: Infiltrating every mobile phone by 2010?”*
gadgetell.com
http://www.gadgetell.com/tech/comment/no-escaping-rfid-infiltrate-every-mobile-phone-by-2010
(Published 2009)

[57] Ö. Eraslan, *“Enhancing Security and Usability features of NFC”*
School of Computing, Blekinge Institute of Technology
(Published 2009)

[58] M. Koenig, *“Java Card 3 Programming”*
Smart University
http://www.michel-koenig.eu/javacard/javacard3.pdf (overview)
http://www.michel-koenig.eu/javacard/td3.pdf (encryption using Java Card 3)
(Lookup 2011)

[59] Common Criteria (CCRA) homepage
http://www.commoncriteriaportal.org
(Lookup 2011)

[60] Niwano, Durand, Gaston, Faher
“Open Smart Card Infrastructure for Europe v2”
eESC TB7, Europe Smartcards,.
http://www.page9.webaxxs.net/iosis/oscie/Download/05-4.PDF
http://www.eeurope-smartcards.org
(Published March 2003)

[61] Google Wallet
http://www.google.com/wallet/how-it-works.html
(Lookup 2011)

[62] *“Wireless carriers create new mobile payment network”*
Wireless industry news
http://www.wirelessindustrynews.org/news-nov-2010/2218-111710-win-news.html
(Published Nov 2010)

“MasterCard approves Gemalto's new NFC payment software on a SIM”
Wireless industry news
http://www.wirelessindustrynews.org/news-mar-2011/2454-033011-win-news.html
(Published March 2011)

[63] *MIDlet*
Wikipedia
http://en.wikipedia.org/wiki/MIDlet
(Lookup 2011)

[64] Nicola Dragoni, “Security-by-Contract for Applications’ Evolution in Multi-Application Smart Cards”
Technical University of Denmark (DTU)
http://www2.imm.dtu.dk/courses/02234/slides/Security_smart_cards.pdf
(Lookup 2010)

8 Terminology and abbreviations

Abbreviation	Description
3GPP	3rd Generation Partnership Project [3]. European collaboration of telecommunication associations. Evolves the GSM standard for third (and future) generation mobile phone systems. The 3GPP standards 03.19, 11.11 and 11.14 relate to SIM cards and GSM cellular phones and are of particular interest to developers.
AID	Application Identifier. Identifies an application on a Smart Card. Defined by the ISO 7816-5 standard. This is split in two parts, first 5 bytes is registered application provider (RID) which is typically the software vendor ID and a 11-byte proprietary application identifier extension (PIX) that identifies the application for this vendor.
APDU	Application protocol data unit (defined by ISO 7816). These are the command protocol used by communication between SIM and Mobile phone. They are encapsulated inside a TPDU.
ASN.1	Data encoding rules. Used for example when defining communications protocols. Governed by ISO/IEC 8825-1 BER: Basic Encoding Rules, CER: Canonical Encoding Rules, DER: Distinguished Encoding Rules.
ATR	Answer to reset (initiated by smart card's hardware reset pin). Up to 33 bytes are sent back. Contains supported transport protocol, maximum data transmission rate and various hardware parameters.
BER	Basic Encoding Rules (according to ASN.1). Some commands can have the APDU body consisting of multiple "tags". BER defines how these tags are coded. Each tag is typically quite short.
BIP	If the SIM card is BIP compliant it means the SIM card can communicate with a server using the phone's GPRS or 3G as a carrier.
CAD	Card acceptance device. Basically a smart card reader. It sits between the SIM card and the phone/PC/cash register and so on. It is also called a *terminal*.
C-APDU	Command (request) APDU.
CAT ET TS 102 223	Card Application Toolkit (here also called *STK*). "CAT Application" here means an *STK Applet*. SCP/CAT is defined by standard ETSI TS 102 223 / 3GPP TS 11.14. Among other things it provides details about the APDU commands described in Appendix C.
CDMA	USA-based 2G mobile standard. Standardized by the TIA-USA organization. Many CDMA standards do not use the replaceable smart-card solution, meaning the phone number is tied to the phone hardware.

CDMA2000 (CDMA/3G)	Evolved CDMA standard, extending it to support 2.5/3G telephony. Competes with UMTS.
CLA	Class byte (see Appendix B).
CSIM (CDMA)	CDMA Subscriber Identity Module (actually an application running on a R-UIM). It contains phone number, encryption keys, SIM serial and other information needed to connect to and use a CDMA network.
DF	Dedicated file; directory in file system.
DIR	Directory.
EF	Elementary file; standard file in file system.
EMV	Stands for "Europay, MasterCard and VISA". Smart Card standards for electronic payment. EMV Follows the ISO 7816 standard (Note that ISO 7816-4 commands are common for GSM SIMs and EMV smart cards). EMV card RIDs starts with "A00000.."
ETSI	European Telecommunications Standards Institute [2]. ETSI produces globally-applicable standards for Information and Communications Technologies. Among many other things they govern GSM and SIM specifications together with the 3GPP.
FCI	File control information (See file system section)
FCP	File control parameter (See file system section)
FID	File identifier, two byte identifier for an element of the SIM file system.
FMD	File management data (See communication section)
GlobalPlatform	Organization for establishing standards for Smart Cards (UICC).
GSM	Global System for Mobile communications (GSM comes from French "Groupe Spécial Mobile"). Originally European based standard but used world-wide. Competes primarily with CDMA (USA) and localized standards in China and Japan. The GSM standard is since 1989 managed by the *ETSI*.
ICC-ID / SSN (GSM)	SSN (Smart card Serial Number) is unique and consists of 19-20 digits. Since a SIM follows the ISO/IEC 7812 this is also equal to the chip's ICC-ID number as dictated by ISO/IEC 7812.This number can be found printed on the physical SIM card, where the number always starts with ***89*** to denote it belongs to the telecommunication industry (others are reserved for banking etc.). It is followed by vendor ID, country code and an account id and ended with a parity check digit.
IEC	International Engineering Consortium [4]. Group of universities and engineering societies. Works together with ISO to provide world-wide standards.
IMEI (GSM)	A GSM mobile phone's unique hardware serial number. Used among other things to identify phone in network and to block stolen phones.

IMSI (GSM)	International Mobile Subscriber Identity. The unique identifier (serial number) for a SIM card. The first digits deduce the SIMs origin country and operator followed by an operator unique serial number.
INS	Instruction byte (see Appendix B).
ISO	International Organization for Standardization [1].
ISO 14443	ISO standard for contactless cards (NFC/RFID).
ISO/IEC 7812 (GSM/CDMA)	Standard for magnetic swipe cards and the hardware and communication protocol specifications for the chip found on credit cards. A UICC is made following this standard.
ISO-7812 ISO-7816	The primary ISO smart card standards. Mainly 7812 governs physical layout/electrical connections and 7816 the command protocol.
ITU	International Telecommunication union. Special agency of United Nations dealing with telephony and radio communications. Both UMTS and CDMA2000 standards are endorsed by the ITU.
Java Card	Framework developed by SUN Microsystems for enabling "Java" applications to execute on a smart card. It provides a hardware independent platform.
JCRE	Java Card Runtime Environment (JVM plus libraries)
JCVM	Java Card Virtual Machine (limited JVM)
JVM	Java Virtual Machine
Ki (GSM)	Network authentication/signing key, 128 bits. This key is assigned by the operator and stored both in operator's database (the *Authentication Centre*) and on the SIM by the issuer.
LAI (GSM)	Local Area Identity. Part of the TMSI. Updated (pushed) by the network to the SIM. LAI is country code (CC), mobile network code (MNC) and a local area code (5 digits).
ME, MS, Modem, Terminal	ME stands for *Mobile Equipment*, in this text this is a typically meant a cell phone (handset). In technical documentation it is also called a Modem, Terminal or MS (Mobile Station).
MF	Master file; root in file system, (See file system section)
MF	Instruction byte
MSISDN (GSM)	Cellular phone's number in GSM networks.
MVNO	Mobile virtual network operator. Such an operator does not own a network but leases it from some other operator. MVNOs only supply customers with SIM-cards. Examples of such operators in Sweden are Hale-bop and Tango.
NFC	Near Field Communication [53]. Wireless communication between SIM and external device over short distances. Makes it possible to

	easily authenticate purchases in shops directly on your phone. Typically the NFC unit is a hardware part of the phone and not integrated directly on the SIM (other smart cards with on-board NFC can be directly powered by the radio waves). See ISO 14443 and 18092.
OMA	Open mobile alliance
OTA	Over-the-Air (usually meant for protocols implemented over a protocol implemented in the cell phone, communicating over GPRS/3G or distributed through multiple SMS).
P1-P2	Parameter bytes (See communication section)
PIX	Proprietary application identifier extension. 11 bytes field that identifies the application from the vendor specified by the RID.
PKI	Public Key Infrastructure (usage and storage of public and private keys, encryption and so on)
PTS	Protocol type selection (used by reset protocol)
R-APDU	Response APDU (response to a C-APDU).
RFID	Radio Frequency Identification. Most interesting about this technology is that the radio waves of the transmitter can power the smart card incorporating the RFID technology. This is used for example with intelligent identification cards / door openers, animal IDs, libraries, tracking cargo shipments and much more. See ISO 15693 and *NFC*.
RFU	Means “Reserved for Future Use”
RID	Registered Application Provider. This is a unique 5 byte value identifying a Smart Card application’s vendor. It is the definition of the first 5 byte portion of an AID.
R-UIM (CDMA/GSM)	Removable User Identify Module, the CDMA network analogue to a SIM-card. It is based on the GSM SIM specification/hardware and can be used in a GSM phone (enabling its GSM part only). R-UIM has been replaced by the CSIM and UICC standards.
SCP	Smart Card Platform (defined by the ETSI/3GPP “SCP” documents)
SCP	Secure Channel Protocol. Smart-card encrypted communications protocol defined by GlobalPlatform.
SIM (GSM)	Subscriber Identity Module. Provides network identity in GSM/3G networks. SIM cards are also used by many satellite phones.
SM	Secure messaging, meaning that the message is encrypted.
SMS	Short Messaging Service, consisting of 160 7-bit characters or 140 8-bit bytes. International characters are encoded in the Unicode UCS-2 character set (2 bytes/character, 70 maximum).
STK / SAT	Here; SIM Application Toolkit. Governed by ETSI TS 101 267 / 3GPP TS 11.14
SW	APDU response status word (See communication section). This is

	the response code for an executed command. It is composed of two bytes (SW1 and SW2). 0x09000 (and 0x09100) means "success".
TIA-USA	Telecommunications Industry Association. America based organization evolving the CDMA and CDMA2000 standards. As a curiosity they also standardized the basic HAYES command set (for modem control).
TLV	Tag length value (See communication section)
TMSI (GSM)	Temporary Mobile Subscriber Identity. Basically identifies the cell the mobile currently is in. Updated as needed.
TPDU	Transmission protocol data unit, defined by ISO 7816-3, most used types are: T=0 : Byte oriented, T=1: Block oriented, T=CL, contactless protocol for RFID/wireless cards, T=USB for USB and T=RF for near field communications.
TS	TS stand for *Technical Specification* (refer to ETSI TS …)
UCS2	Older encoding for 16 bit Unicode. Used by the GSM standard. http://en.wikipedia.org/wiki/UCS2
UICC (CDMA/GSM)	Universal Integrated Circuit Card. Modern design, making it possible that a single card can simultaneously host applications/logical extensions for SIM, CSIM [for *CDMA*] and USIM [for *UMTS*]. The UICC standard is governed by the ETSI.
UMTS (GSM/3G)	Universal Mobile Telecommunications System. 3G (soon 4G) which typically uses W-CDMA as a carrier. Standardized by the 3GPP (Europe).
USIM (GSM)	Universal Subscriber Identity Module. Extension of the SIM standard for 3G (UMTS). This runs as an application/logical extension on UICC compliant smart-card hardware. The USIM standard is governed by the 3GPP.
USAT	SIM Application Toolkit for USIM. See ETSI TS 31.111.
USSD	Unstructured Supplementary Service Data. Direct messaging protocol between operator service provider and a cellular phone (can be used to create a directly-connected GUI on the phone/WAP services). http://en.wikipedia.org/wiki/USSD
VAS	Value-added service. This can be applications running on the SIM card as well as applications running on the phone, providing services such as ticket purchasing, call-back services and much more.
WIM	Wireless Identity Module, standard how to use PKI (Public Key Infrastructure) on smart cards, based on PKCS#15.
W-SIM (GSM)	Big-sized SIM card with radio transmitter built in. Mentioned as a curiosity only

Table 4: Abbreviations used throughout the document

Appendix A - Timeline for the smart card evolution

This is a short list of historical events in the smart card evolution coupled with what happened for the GSM/3G and Java Card standards.

Year	System	Organization / Company	Description
~1970	Various	Various	Invention and idea of smart-card formed. Still ongoing patent debate who invented it first.
1977		Honeywell bull	First microprocessor smart card.
1981		Various	First generation of smart cards (credit cards) [13].
1987-89	ISO 7816-1,2,3	ISO	Physical card layout, electrics, protocols for smart cards, ISO 7816 standard [27] [28].
1989	Carte Bancaire B0'	Groupement des Cartes Bancaires	First standard for payment cards (predates EMV).
1990	CAVIMA	RD2P, Research & Development on portable files (France).	"CArd VIrtual MAchine" [13]. First virtual machine on a smart card.
1990	J-Code	Digicash (Netherlands)	Byte-code interpreter (not Java) on smart card, concurrent work to RD2P.
1992	CQL	GemPlus	Extending CAVIMA project. Work leads to the ISO 7816-7 standard (Structured Card Query Language, SCQL)
1993	ZC-Easy	Zeitcontrol	C and Pascal derivate
1993	Treff	Europay	Treff (Forth derivate)
1995	ISO 7816-4	ISO	TPDU protocol (smart card request/responses). Security, file system on card standard [27] [28].
1995	EMV 2.0	**E**uropay/**M**astercard/**V**isa	EMV standard based on ISO 7816. New versions came 1996, 98, 2000, 07 and 2008.
1995	Multos	National Westminster, Maosco	Multi-application smart card [24] with application memory separation, virtual machine originally programmed using a Pascal derivate. Today it uses a language-agnostic virtual machine (C/Assembler and Java are supported).
1995 (2004)	ISO 7816-5	ISO	Routines put in place for how to register international RID /AID ranges.
1995	SIM Module	ETSI	Smart card standard for SIM module.

	GSM 02.17		
1995	SIM Module GSM 11.11	ETSI	GSM Standard for SIM module.
1996	Java Card version 1.x	SUN Microsystems and Schlumberger	First Java Card [14] implementation using a subset of the Java byte-code. Executed by a 4Kb big virtual machine. This was the real break-through for virtual machine based cards, finally embraced by the GSM community.
1996	SIM Tool-kit GSM 11.14	ETSI	Specification for SIM application toolkit. Contains APDU commands for SIM/ME.
1996	EMV	EMV Group	Standards for secure payment on payment terminals (those found in shops) to read smart card module on VISA cards and use PIN authorization instead of magnetic card swipe and signatures.
1997	JCOP	IBM	IBM's Java Card implementation [17], JCOP is their smart card operating system, now maintained by NXP / Philips.
1997	Java Card version 2.0	SUN Microsystems	Huge changes from version 1.0, introduced Java-Card Applets. Still did not specify the application's binary format.
1998	Java Card GSM 03.19	ETSI	SIM API for Java Card (APDU/Proactive commands etc.).
1999	GlobalPlatfo rm	GlobalPlatform	GlobalPlatform group founded to continue developing VISA's Open Platform standard.
1999	USIM 3G TS 21.111 v3.0.0	3GPP group	Universal SIM. Advanced SIM for 3G/UMTS.
1999	Java Card 2.1.x implementati on; 32k	SUN Microsystems	Released version 2.1 of specifications for JCRE. Binary format of Applet and Applet-loading procedure was standardized. Added support for "shareable" interface objects that can be accessed from multiple applications. The first Java Card 2.x SIM was developed.
2000	UICC Smart Card Platform	ETSI / SCP	Universal Integrated Circuit Card. Advanced Smart Card standard. Newer SIM cards are all based on this standard.
2000	S@T	Gemplus, Giesecke & Devrient (G&D), ORGA Kartensysteme and Schlumberger	Standard for dynamically downloading Java Applets OTA onto the smart card to provide operator service portals (where they can provide value added services). Original S@T only provides WAP / Text only

			services and communicates over SMS (!).
2002	RUIM-Card	3GPP	UICC for CDMA.
2002	Java Card 2.2	SUN Microsystems	Four logical channels to provide simultaneous transactions between SIM and ME (2.2.2 has 20). Initial support for contactless cards.
2003	Gemplus(Gemalto) / Univ of Lille	CAMILLE	Real time smart card operating system [13]. Has Just In Time (JIT) compiler to provide fast Applet execution
2005	ISO 7816-4 (extended)	ISO	Smart card physical interface extended to include radio and RFID communication. This means it is possible to communicate with a smart card without having to insert it into a physical reader.
2006	UICC / Global platform v2.2	GlobalPlatform / ETSI / EMV (Visa etc.). First known as *Open Platform*.	De-facto standard for UICC cards, involves software and hardware minimum requirements for multi-application smart cards, also standardized Over the air (OTA) application deployment for SIM.
2006	Java Card 2.2.2	SUN Microsystems	More encryption algorithms (AES, elliptic curves) and better memory management (such as optional garbage collection). This is the Java Card standard that is most commonly found in SIM cards on the market today (2010).
2007	USAT (Extended S@T)		Extended S@T (graphics and so on) uses special Wireless Markup Language (WML) formatting language. Provides encrypted pages (WIM).
2007/8	Dot Net platform	Gemalto / Microsoft	SIM card Dot-net security platform
2007/8	NFC/RFID	ISO / IEC / ETSI	NFC and RFID standards [53]. First mobile with NFC support appears 2007.
2008	Java card 3.0	SUN Microsystems (Oracle America Inc)	Java Card 3.0 [14] platform for 32-bit CPUs with multithreading, implements garbage collection, strings, 32-bit data types, servlets (HTTP) etc.
2009	LTE UICC	Gemalto and others	Java Card 3 enabled SIMs for LTE and 3G networks.

Table 5: Historical evolution of smart card standards and platforms.

Appendix B - Communicating with the smart card

Hardware interface

When the mobile phone (or external card reader) communicates with the smart card it is done over a serial bus (defined by ISO 7816-1/2) [27] [28]. Newer smart cards can make use of USB, USB2 or even NFC (see section 1.5), but the standard is a simple 9600 baud serial line (some more modern cards can change this post-reset to use up to 115kbps). The serial pin can either be configured to be input or output, this mean that data cannot be sent in both directions at the same time. If you want to use a computer to communicate with a smart card you will need a smart card reader, this is typically plugged into the computers RS232 or USB port.

Most such smart card readers use a credit-card sized docking bay for the smart card, there are adapters where you can place a small scale factor SIM that then fits into the docking bay. When you get a new SIM from the manufacturer it usually comes encapsulated into a credit-card sized piece of plastic, you can then break the SIM loose after you have finished imprinted it so you can insert it into your phone.

ISO 7816-3, low level transmission protocol and TPDU

This ISO standard defines the default serial protocol used by smart cards. It includes configuring serial bus speeds, start/stop bits and the rest needed for establishing serial communication. It also defines the higher level protocol to send request and responses back and forth between the smart card and the ME (can be a smart card reader, phone, set top box etc.). This protocol uses TPDU (Transmission protocol data unit) as the basic data blocks. The standard specifies that the smart card is the slave unit meaning it is the ME that sends requests (commands) and the card is returning responses.

Protocol types

Protocol T=0 (default)

A character based, asynchronous half duplex protocol. This is the default protocol [29] after card reset.
It uses parity bit on each byte. Low level ACK/NULL flow control bytes are also sent.

Protocol T=1

A block based protocol. It uses CRC, block types and other techniques to handle data sent in blocks.

Protocol T=USB

Used for USB communications.

Protocol T=RF

Used for radio communications (contactless cards protocol standard).

TPDU formatting (Transmission protocol data unit)

A TPDU consists of a header and a body. The layout is like the following.

Offset	Name	Description
0	CLA	Instruction class (**FFh** selects protocol to use)
1	INS	Instruction code
2	P1	Instruction parameter 1 (if P1 and P2 are used as one 16 bit value then P1 represents the MSB and P2 the LSB).
3	P2	Instruction parameter 2
(4)	(P3)	Number of data bytes which should be transmitted in the body of the TPDU. For requests value 0 means 256 bytes and for responses it means 0 bytes. This byte does not exist for all CLA combinations (to save space)
(5..)	Body	Optional data body.

Table 6: TPDU formatting.

ISO 7816-4, command/response protocol, APDU

An APDU (Application Protocol Data Unit) is the data item that is visible on the application level. It consists of a header and a body, just like the TPDU. In fact normally the APDU and TPDU packets are interleaved, they use the same header encoding standard (certain set of commands are used only by TPDU and APDU respectively). This style of protocol is not following the OSI protocol's layered design but was done to conserve space (and data transfer times).

The signaling / flow control and block assembling/disassembling (enveloping) is handled by the TPDU layer. Note that in card emulators and development software the TPDUs carrying the APDU have typically already been assembled / disassembled for you, so what is shown in "APDU logs" and similar are the final APDUs not the intermediate TPDUs.

An APDU block can be quite large (assembled of several TPDUs). The maximum APDU data body size is 64k, which is fine since it will take several seconds to transfer if done over the serial port at the standard speed. The 64k limit was extended in recent SIM standards, R5 and above which also provide faster communication speeds.

APDU transactions

An APDU message consists of a header followed by an (optional) data body. The command (C-APDU) and response APDU (R-APDU) are formatted differently. The APDU data body in turn can consist of smaller sub-entities. The initiator party sends a command (C-APDU) that might be split and enveloped by one or more TPDUs.

The recipient assembles these parts, parses the assembled APDU and responds with a response (R-APDU) in similar manner (response can be split into multiple TPDUs). Integers bigger than a byte are transferred in big-endian format, for example the 16 bit number 1234h is sent as byte 12h (offset 0) followed by byte 34h (offset 1).

Logical channels

Java Card version 2.2.2 introduces the concept of asynchronous messaging, where requests and responses can be interleaved. This is very practical for messaging that takes time to process and is needed to be able to execute more than one application at the same time on the smart card (no need to *deselect*).

Logical channels are used to send APDU commands between the phone and a particular smart card application. There is just one physical channel, which is the serial interface, but there can be multiple logical channels; which one to use is defined in the APDU request inside the CLA byte. For example, this makes it possible to read and write to multiple files at the same time, since the SELECT command is done for the given channel.

Encoding of APDU and TPDU commands

Like previously described, the APDU and TPDU uses the same header layout but some CLA (class of instruction) combinations are reserved for TPDU and the rest for APDUs. This means the TPDU/APDU protocols are interleaved. The basic header format has already been described earlier but is given here in more detail.

The APDU command header

Base header consists of the following 4 bytes (values given in hexadecimal for clarity).

- **CLA (Class of instruction)**
  10-7Fh: Reserved for the future (RFU)
  0X,8X,9X,AXh: *Command, optionally secure (most are standard ISO-7816).
  B0-CFh: Commands (standard ISO-7816).
  D0-FEh: Vendor specific commands
  FFh: PTS (TPDU protocol type selection, see TPDU section).

  The *command* class (*) have security attributes and used logical channel encoded into the CLA byte, using the lowest four bits (thus the “X” above). These four bits are encoded as follows:
  Bit 0, 1: Logical channel (up to 4 concurrent commands can thus be executing).
  Bit 3=1: Use standardized secure messaging (SM) according to the ISO standard (clause 1.6). If bit 2 is set it means the header is authenticated, if it is clear it is not.
  Bit 3=0: No secure messaging (Bit2=0) or use card vendor proprietary encoding (Bit2=1).
- **INS (Instruction code)**
  See APDU instruction codes section below. TPDU instruction codes are not explained here.
- **P1 (Instruction parameter 1)**
  If not used by the instruction it must be set to 0.
- **P2 (Instruction parameter 2)**
  If not used by the instruction it must be set to 0.

The P3 byte is not seen by the APDU level, it is used by TPDU only to be able to assemble/disassemble an APDU into multiple TPDUs.

The APDU command body (optional)

This contains the following components, note that not all fields might be present; this is defined by the header:

- Lc field (0-3 bytes); gives number of bytes in the Data field.
- Data field (containing APDU request data).
- Le field (0-3 bytes); Gives the maximum allowed number of bytes in expected command response. 0 means give maximum possible number of bytes back.

The encoding of the length fields (Lc, Le and Lr) is quite complex and a full explanation is outside the scope of this thesis (for more information see section 5.3.2 and Annex-A of ISO 7816-4). It is meant to pack the length encoding data in an efficient way to save space. In short a single byte of 01-FFh means length 1-255 and a three byte sequence of 00h, high-byte, low-byte are used for lengths 256-65536 (65536 encoded as both low and high bytes set to 0).

The APDU response

As usual the implementer of the standard tries to pack data as efficiently as possible. Thus the format of the R-APDU (response) depends on the C-APDU (command). These are the four APDU request/response combinations.

Case	Command (Request)	Response fields expected
1	Header	SW1, SW2
2	Header, Le	RESPONSE DATA BODY, SW1, SW2
3	Header, Lc, Data	SW1, SW2
4	Header, Lc, Data, Le	RESPONSE DATA BODY, SW1, SW2 For this case the response is typically split into two TPDUs

Table 7: APDU response formatting.

The two trailing bytes SW1 and SW2 form the Status Word (SW).
- SW1 (high byte of response code, SW)
- SW2 (low byte of response code, SW)

The Response Data body is a data block with a maximum allowed length given by the command's Le field. If the length of the returned data cannot match the given Le field exactly an error code (in SW1/2) will result. This indicates the recipient needs to do additional handling in order to retrieve the returned data. This process quite complex, see Annex "A" of ISO 7816-4.

APDU responses codes (status word)

The listed codes here are just a short subset of all of the defined result codes. See ETSI TS 102 221 section 10.1.2. Mostly you will see "90 00" which means success.

SW	Description
9000h	Command successful
6F00h	Unknown result
6D00h	Command not allowed (invalid/unknown/unauthorized instruction)
67xxh	Length fields wrong, cannot recover, response lost (value of "xx" does not matter)
61xxh	More (xx bytes) data is remaining to be read
6Cxxh	Le not accepted. Actual available response length is called "La" and is contained in "xx". Command can be re-issued in order to retrieve the data.
91xxh	Command successful, there are also proactive commands awaiting execution (ME should use FETCH).
92xxh	Various memory errors
94xxh	Various file errors.
98xxh	Various security errors.
9Fxxh	Success, xx bytes available to be read with "Get Data".
6A82h	File not found
6A81h	Function not supported
6283h	Selected file is deactivated

Table 8: APDU response codes.

APDU data body formatting

Most APDU commands need multiple blocks of data arguments to function (for example a text block needs length, encoding and actual text data). Such blocks of data are called *tags* and are placed in the APDU's data body. Tags with bytes starting with **00** and **FF** are simply ignored (padding of areas previously covered by deleted objects). The layout of these tags follows either of the two standards *BER-TLV* and *SIMPLE-TLV*. TLV stands for *Type-Length-Value* (three components). BER stands for *Basic Encoding Rules*. All of these are defined by the *ASN.1* standard (ISO/IEC 8825).

BER-TLV data objects (GSM spec 11.14 section 6.5)

These represents multiple tags contained in one APDU body. Each BER-TLV data object consists of 2 or 3 consecutive data fields. The tag field can denote security class/type or number, followed by a length field followed by a value field. The body data contains a number of SIMPLE-TLV objects.

SIMPLE-TLV data object (GSM spec 11.14 section 11)

These has a simpler layout than BER-TLV objects. Tag field is a single byte (1-254) followed by a value length field (1-3 bytes) followed by a value field.

For more information see GSM 11.14 specification and [45], also check the source code for the *sendSMS* method in Appendix F.

Example of an APDU command - SEND SHORT MESSAGE

As said the composition of an APDU command is a header followed by a body containing various sub-tags. Some of these are optional and some are not. The most complex APDU message used by the example application is *SEND SHORT MESSAGE* used to send SMS via the SIM card. All of the basic APDU messages are described by ETS 102 223. The *SEND SHORT MESSAGE* is described in section 6.4.10 (description) and 6.6.9 (header). The command header general format is defined in ETSI 102 221 section 11.1.1.3 and in ETSI 102 223 Annex C.

For SEND SHORT MESSAGE the command header looks like the following:

Description	Clause	M/O/C	Min	Length
Proactive UICC command Tag	9.2	M	Y	1
Length (A+B+C+D+E+F+G+H+I)	-	M	Y	1 or 2
Command details	8.6	M	Y	A
Device identities	8.7	M	Y	B
Alpha identifier	8.2	O	N	C
Address	8.1	C	N	D
3GPP-SMS TPDU	8.13	C	Y/N	E
CDMA-SMS TPDU	8.71	C	Y/N	F
Icon identifier	8.31	O	N	G
Text Attribute	8.72	C	N	H
Frame Identifier	8.80	O	N	I

Table 9: Description of SEND SHORT MESSAGE (from ETSI 102 223 clause 6.6.9)

Description: Field description.

Clause: Where to find documentation about the field inside the ETS 102 223 document. For example clause 9.2 contains a list of the command tags. Luckily the framework provides constants for these (PRO_CMD_SEND_SHORT_MESSAGE in this case). As mentioned earlier tag values 00 and FF are used for padding.

M/O/C: Tells if the field is **M**: Mandatory, **O**: Optional or **C**: Conditional (depends on other fields).

Min: The *minimum set* column defines if the field is necessary for the ME to understand the command. “Y/N” means it depends on some other field’s presence; here either 3GPP or CDMA TDPU has to be present for the command to be valid

Length: Byte length of the field. Often it depends on sub-items inside the field and is referred just as a letter. Actual length is then explained in the fields given *clause* section.

The method *sendSMS* in the example code (Appendix F) fills in the following fields:

- *Proactive UICC Command Tag*
- *Command Details*
- *Device Details*
- *Alpha Identifier* (a text shown on the display of handset during sending)
- *TPDU tag*, a complex field containing things like phone number of destination (as 4-bit packed BCD), SMS class, message encoding and the message text itself. The method packs the byte array containing the text into packet 7 bit ASCII (160 characters per SMS).

As seen there are many more options possible with this command by defining more fields. A detailed study of the standard is necessary when you develop SIM applications; it is by far the most important document for an application developer to read and understand. Further analysis of the tag structures and their contents are outside the scope of this thesis but everything you need is contained in said document.

Appendix C – Smart card file system and security

It is very important to have high *data integrity* on the card and avoiding data corruption. The operating system contains special measures to keep data integrity on the card, even if the card loses power in a write operation. As important is the *data security*, only authorized parties are able to read and write data to the card. Very sensitive information (like encryption keys and the operating system) is typically stored in ROM memory which is designed to be *very* hard to probe from the outside without destroying the chip.

A smart card stores most non-volatile information in a file system. The file system uses EEPROM (or later, Flash) type memory which is slow to write and wears out. The operating system's memory manager on the card has to keep track of bad pages and the page copying that has to be done before clearing an old page, it is vital that no information is lost even if power is turned off while data is being moved. The internal file/memory block system structures also have to be kept in nonvolatile memory which further slows these operations down.

Typical maximum size of nonvolatile memory in a SIM card (R99/R4/R5) used for GSM is around 32k. This is enough to store a phone book and some SMS messages for the user while still being very cheap to purchase for the operator. The file system on a card is governed by the ISO standard 7816-4 [28]. It has the classic tree structure with a root directory, subdirectories and files. It consists of the following elements:

File type	Name	Description
MF	Master File	File system root directory
DF	Dedicated File	Directory file (can contain DF and EF elements)
EF	Elementary File	Data file

Table 10: File system element types.

Each file type has a two byte (unsigned short) file identifier (FID), in this document they are given in hexadecimal notation. This number must be unique in respect to the element's father node.

Two identifiers are used for special purposes and cannot be a MF/DF/EF id, these are:
3FFFh: File path selection (*referencing by path*), see the "Locating a file" section.
FFFFh: Reserved for future.

File tree structure

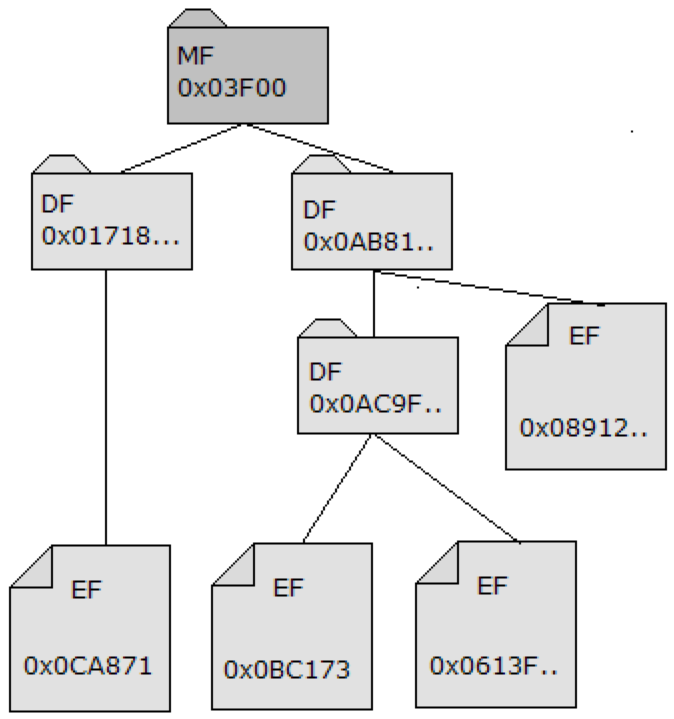


Figure 22: File system components example.

Master File (MF)

Root directory, the top node in Figure 22. Its identifier is always **3F00h**. It can contain DF and EF child elements. There should be only ***one*** file with this identifier in the whole file system.

Dedicated File (DF)

Directory file (DIR node), it can contain DF and EF child elements. It has a Name which is 1-16 bytes (2-32 hex digits) which typically describes an Application ID (AID) as defined by ISO 7816-5. The name must be unique for the current card. Of special interest is directory **7F20h** which contains GSM parameters and **7F10h** various telecom parameters (in EFs) for SMS/MSISDN and so on.

Elementary File (EF)

Standard file (leaf in the file system), it cannot contain any child elements. A file has an extra *short file identifier* field of 5 bits that can be used when locating a file with the SELECT FILE command (see section below).

List of some well-defined file identifiers (FIDs)

0000h : PIN and PUK #1
0100h : PIN and PUK #2

0001h : Application keys
0002h : Manufacturing info
0003h : Card ID info
0004h : Card holder info
0005h : Chip info
0011h : Management keys

Elementary file types

There are two types of EFs; *Internal* (file used by OS, containing settings etc.) and *Working* (used by applications).

Elementary file access

A file can be marked as shareable, which means it supports concurrent access on different logical channels (see section *applications*).

Elementary file storage types

EFs can carry two main types of data structures are defined in the ISO 7816 spec: ***Transparent*** (byte stream) and ***Record*** (sequence of records). The file size can be static or variable. The card must support at least one of: Transparent, Linear with fixed record size, Linear with records of variable size or cyclic with records of fixed size.

- *Transparent files.*
  Byte streams, accessed with READ BINARY / WRITE BINARY. Data is accessed using a file offset, just like a normal file system.
- *Record files.*
  Each record can either be fixed size or be variable. They can either be organized as a sequence (linear file) or ring (cyclic, N positions where N is fixed). Each record has a byte ID with value 01h – FEh (00h is current position and FFh has special meaning).

File header information

Each file has metadata attached; the metadata is formatted according to the BER-TLV specification. This metadata contains DF name, the two byte file identifier, file descriptor byte (defining the file storage type, see section above), file size, security attributes and more. The metadata is arranged as key/value pairs, key is a byte between 80h and 9Fh [28].

Selecting a file

The file system works by first selecting a file (see the SELECT FILE command) in the file system that you will work with for the current logical channel. There are three ways to identify a file:

1. Directly by the two byte file-ID.
In this case the file-id has to be unique among all children in the current directory node (the OS recursively searches among all child-nodes of the current directory).

2. By using a short (5 bit) ID.
Value 0 means currently selected EF FID.
Value 1-30: Select file using short ID 1-30.

3. By using a file-path.
It is possible to locate a file by using a file path (a string of 2-byte file-IDs ordered from MF to EF/DF). For the first ID (FID) in the path the special value of **3FFFh** can be used. This means the path starts relative to the "current directory" of the file system (same as the "dot file" in Linux and windows). Note that short FIDs (mentioned above) cannot be part of a file-path.

4. Referencing by DF name.
Directories (DF) can be selected using its unique name (1-16 bytes).

Smart card file security

There are three main areas of security authentication levels on the card:

- Security status obtained by verifying a password (typically after selecting a file with the SELECT FILE command).
- Security status using a key (attached to MF or DF).
  Authentication is done either with the VERIFY command or the GET CHALLENGE command followed by the (EXTERNAL) AUTHENTICATE command.
- Data authentication (checksums or digital signatures).
- Data enciphering (key management and data concealment with XOR).

ISO 7816-4 instruction codes for security and file access

The most important instruction codes are listed here, it also gives an idea how things actually works. See the ISO 7816-4 for details (BER-TLV encoding for command data arguments etc.).

INS (hex)	Command	ISO 7816-4 clause	Description
0E	Erase binary	6.4	Deletes portion of a binary file.
20	Verify	6.12	Unlocks a file/directory on the SIM for access

			by providing a password. This password can be unique per file/directory. Unsuccessful attempts might lock the resource on the card.
70	Manage channel	6.16	Open/closes/selects a logical channel to the card. At least four channels are available.
82	External authenticate	6.14	Authenticates an issued challenge sent by the terminal (see “Get challenge”, both terminal and card uses this data). A successful result means the selected item (EF/DF etc.) is unlocked for use by the terminal. Unsuccessful result might mean that the item is locked; the card typically keeps an attempt counter.
84	Get challenge	6.15	Asks the card to generate a challenge token for either the selected EF/DF or for a specific secret key (up to 31 per card) and send the result back. Receiver does computations on this token and then issues an External Authenticate command and attaching the result. If the data matches the selected file (EF/DF etc.) is unlocked.
88	Internal authenticate	6.13	Authenticates the card or a particular directory with the terminal (meaning the terminal can now trust it). This is done by letting the card compute authentication data (the “challenge” given by the terminal) and returns it back to the terminal, which can compare it to a locally encrypted version of the same data. This basically checks that the key on the card and on the terminal is the same without directly comparing the key data. Typically the key on the card is given implicitly by using SELECT of the MF (root directory) of the card or a DF (standard directory). This means that the terminal can trust either the contents of the card as a whole or of a particular directory. For some cards it is possible to select both algorithm and key to use with INTERNAL AUTHENTICATE using other commands not described here. The card can keep track of number of attempts to restrict the usage of the particular key and/or algorithm.
A4	Select file	6.11	Selects (and authorizes usage of) a file for the

			current channel. All following read/write commands are done towards this file.
B0	Read binary	6.1	Reads binary data from byte oriented file. Offset from beginning of file and size to read is given.
B2	Read record(s)	6.5	Reads current or specified record(s) from a record oriented file.
C0	Get response	7.1	Used to retrieve big data responses when long response format (case 4) is not directly supported by the card. These cards returns with result code **61xxh** and full response has to be fetched in blocks with the **Get Response** command. Often such responses are split up transparently into separate TPDUs in the low level protocol handler.
C2	Envelope (files)	7.2	Used to send an encrypted APDU command (including all header bytes, body etc.) inside the data body of the Envelope command. This is needed because the transport protocol (TP) needs to be able to understand the header bytes of the transported data.
CA	Get data	6.9	Gets data from the application running on the card (note that this does not have anything to do with the file system).
D0	Write binary	6.2	Write binary stream to byte oriented file. Apart from writing given data it can also set or clear a number of continuous bytes in the file.
D2	Write record	6.6	Write record(s) to record-based file. Data write operation can be “write once” / logical OR / logical AND.
D6	Update binary	6.3	Overwrites existing data in a binary file. User provides offset and number of bytes to write. This is actually the corresponding command to “read binary”.
DA	Put data	6.10	Puts (sends) data to the application running on the SIM card. Corresponding function is “Get Data”.
DC	Update record	6.8	Updates (overwrites) current or specified record in a record-oriented file.
E2	Append record	6.7	Adds a record to a record-oriented file.

Table 11: APDU commands for file system access.

Hardware level security

The smart card has a number of anti-tampering techniques [58]. The details are kept secret. The chip itself is embedded in an epoxy plastic coating which should make it to break if opened. It is also coated in an x-ray shield to avoid nonvolatile memory scanning. The chip tries to avoid varying the power consumption and answer delay during execution of commands. This is done since some encryption algorithms can become sensitive to intrusion if it can be determined which branches that are executed.

The level of security obtained by a smart card is typically given by the *Common Criteria Level* [59]. The cards are certified and verified by the organization *Common Criteria* (CCRA). Also the applications on the card typically have to be certified which causes problems with post-issued applications. Also see ISO/IEC standard 15408 for more information about this subject.

Appendix D – SIM specific APDU commands

The APDU commands in this section are specific to the GSM/3G standards and are defined by the documents created by ETSI/3GPP organizations. This group of commands is typically not implemented on non SIM smart cards. They are called *Proactive Commands* and refer to the SIM and phone interaction. The document's identity is ETSI TS 102 223 [2] which has its equivalent in 3GPP TS 11.14.

These commands describes how the SIM and phone interacts, for example the SIM can display texts on the phone, create menus, get user input, make a call, send a SMS and even in more advanced SIM cards open a web browser or perform communication over TCP/IP. It is necessary to understand how to build this class of commands to do a functioning SIM card application.

There are very many commands already defined and more are implemented as needs come with increased SIM complexity. For example more recently the SIM can start Web browsing and use TCP/UDP/HTTP transactions directly over 3G networks. Some of these advanced commands are described in the section *networking* below as an illustration how complex the SIM cards of today really are.

In the following sections a number of commands are described. In the s*pecification* column you will see which section in the ETSI document you can find more information about the command. In fact you will spend quite some time to make a command work, some variants and combinations are not supported by all SIMs / phones. You will find this out the hard way even if you get help from the TERMINAL PROFILE command. There are some Java Card library helper functions to construct commands (BER-TLV formatters, various constants, identifiers and bit patterns).

Proactive command layout

- Proactive UICC command tag (one byte).
- Data length of the following data fields (1 or 2 bytes).
- *Command details.*
- *Device identities.*
- *Text string.*
- *Icon identifier.*
- *Immediate response.*

Existence of the items marked in italic text depends on the command type.

SIM/Phone control, proactive commands

The SIM card and the phone interact with APDU messages just like any smart card. However, for smart cards it is always the *Terminal* (here a phone, also called ME or handset) that does a request to the UICC (here SIM card) and then the UICC responds back. A *Pro-Active command* is used when the *SIM card* wants to initiate some action on the handset (displaying message, getting user input etc.). Since the smart card standard had already been set the underlying protocol mechanisms could not be changed. Instead a polling scheme was implemented (STATUS) and additionally all normal APDU commands got extended so they can now return “91 xx” as result code which means “operation was completed plus SIM has additional command for ME to perform” (Result 90 xx is the smart card standard of successful operation).

The phone has also to periodically poll the SIM with STATUS commands to check if the SIM has something for it do. A pro-active command is then received by the ME issuing a FETCH command, the response contains the embedded request to be executed by the ME, when it is finished it issues a TERMINAL RESPONSE command to the SIM containing the result. Since the SIM card tries to remain in power saving mode this means that the status polling shouldn’t happen too often, this fact can cause the user interface interactions to appear a bit sluggish. The polling interval is controlled with the APDU command POLL INTERVAL.

Flow control APDU commands

Command (INS code is given in ETSI section 9.8)	ETSI TS 102 223 clause	Description
Fetch	6.3	As previously discussed normal APDU commands ends with “90 00” being sent back to the phone. However, when the SIM card has something it wants the phone to do it sends back “91 xx”, where xx is the length of the FETCH command containing the embedded APDU to be executed by the phone. If both the SIM card and the phone implement it, it is possible to perform what is called *multi-threaded fetch* where more than one request to the phone might be ongoing at once.
Terminal Response	6.8	The handset (Terminal) responds back with a TERMINAL RESPONSE message when it has finished a FETCH-based request. Note that this can take some time since the user has to perform some action (or the phone was busy doing something else), thus many commands can return a timeout result.
Poll interval	6.4.6	Negotiates how often the phone (terminal) polls the SIM with STATUS commands. This is necessary so the SIM

		can inform the phone that it wants some action to be done (FETCH). This activates polling (Polling on).
Refresh	6.4.7	Reinitializes the phone, can also reset the SIM CPU.
More time	6.4.4	Gives smart card more time to process current command. This is done to avoid lockup of terminal GUI because of lengthy SIM operations.
Polling off	6.4.14	Disables proactive polling (i.e. sending periodic STATUS commands).
Set up event list	6.4.16	The SIM can ask to be notified what the ME is doing by subscribing to events. For example it is possible to be notified when the ME is busy, idle, user is doing something etc.

Table 12: APDU commands for flow control.

Timer handling

When you want some action to be done at a specific interval or time without having your application running you use the timer management command. The SIM can start and stop a timer and also check its current value. Timers can generate events that make a SIM card Applet become active at a certain time.

Specification: ETSI Section 6.4.21 (not displayed in detail here).

User interaction proactive commands

The following table lists the user interaction commands (i.e. the GUI methods).

Command (INS code is given in ETSI section 9.8)	ETSI TS 102 223 clause	Description
Display text	6.4.1	Displays a text message on the phone (with normal or high priority). The text can be specified to have an optional icon (indexed) before it, this looks much like the Windows message box that can contain an Exclamation mark, warning symbol and so on. The message can be selected to be either shown for a short duration or to wait for user response (like back button, OK etc.).
Get inkey	6.4.2	Display text and/or icon (like Display text) and waits for the user to press a single key on the phone. A timeout can optionally be specified.
Get input	6.4.3	Display text and/or icon (like Display text) and waits for the user to enter N digits (possibly including * and

		#) on the phone (a string). Input can be requested to be hidden by stars (used for password entry).
Play tone	6.4.5	Plays a tone on the phone according to type (like Dial, Busy, Error, Ringing etc.).
Set up menu	6.4.8	Used to set up a menu on the phone. The Java Card Library has helper functions so you don't have to do this directly. The user can use the phone's arrow keys to select an item or back to exit the menu, the user's choice is in the phone's response message.
Select item	6.4.9	Selects (moves the cursor to) a menu item.
Set up idle mode text	6.4.22	Sets a message to be displayed when the phone is idle.
Language notification	6.4.1	Informs phone about the currently used Applet language.
Set frames	6.4.35	Create scrollable regions ("windows") on the terminal display. Such a region is called a "frame". This command is not supported by older phones.
Get frame status	6.4.36	Get information about frames defined on the terminal (see above)

Table 13:APDU commands for user interaction.

Note that all texts displayed and/or entered use one of these three encoding types.

- Seven bits GSM character set [36]. Data is packed as tightly as possible into the octets.
- Eight bits GSM character set. One character per byte. This is typically only used to transmit binary data to and from the SIM/phone.
- Sixteen bit UCS2 encoding (limited version of Unicode UTF-16). This is encoded as two bytes (16 bits) per character. A specific phone might not be able to display all of the possible 65536 characters.

Device access and information

The SIM can access some specific hardware on the terminal other than the display and speaker, for example it is possible for an Applet on one SIM card to control other SIM cards in use on the same terminal (multi host). It is also possible to directly execute HAYES commands on the terminals (i.e. the phones) "modem".

Device access APDU commands

Command (INS code is given in ETSI section 9.8)	ETSI TS 102 223 clause	Description
Perform card APDU	6.4.17	Perform APDU command on other SIM card.
Get reader status	6.4.20	Query phone about SIM reader status (if a SIM card is inserted or not in specific slot etc.).
Power off card	6.4.18	Powers off an additional SIM card.
Power on card	6.4.19	Powers on an additional SIM card.
Run at command	6.4.23	Execute Hayes command on phone's modem part.
Terminal profile	5.2	Terminal (phone) returns a number of bytes containing flags describing the phones features; especially it declares which APDU commands that can be used with this card). Currently around 30 bytes are returned but the number is growing for each release.

Table 14: APDU commands for device access and modem interaction.

GSM/3G network communication, messaging and signaling

It is possible to send and receive data through the terminal to other devices in the operator‘s network or on the Internet (GPRS/3G). Here follows a short list of some of the more interesting features. Note that HTML browser support is only supported by very recent SIM cards.

Communication, messaging and signaling APDU commands

Command (INS code is given in ETSI section 9.8)	ETSI TS 102 223 clause	Description
Send short message (SMS)	6.4.10	Sends a SMS. The character set to use is selected, message can thus be fully binary which is typically used when you send data from your Applet to some server machine. The number of characters that can be sent in a SMS is dependent on which type of character encoding used. Seven bits GSM character set gives 160 characters (packed into 140 bytes). Eight bits GSM character set allows 140 characters per SMS. This can be used to send a fully binary SMS, for example containing parts of applications being downloaded (OTA/Over the Air) or parts of pictures. Sixteen bit UCS2 encoding (70 characters per SMS. Also see ETSI TS 123 040
Send USSD	6.4.12	USSD messages are sent directly into the operator’s network and they contain some specified end-point ID, which is typically a server machine that handles the request and sends a USSD response immediately back to the handset. The round-trip is very quick and can thus provide interactive dialogs with some service (WAP). For example pre-paid top-up is done by issuing a USSD command on the handset (Example is *123*1234567#, where 123 is the endpoint-id and 1234567 is the data).
Set up call	6.4.13	SIM can initiate a voice call to the specified number.
Send DTMF	6.4.24	SIM can send “DTMF tones” into the operator’s network; this is basically the same thing as the user entering digits on the keypad of the phone.
Launch browser	6.4.26	Launch a web browser on the phone with the specified URL. Obviously only modern phones supports this (check with the TERMINAL PROFILE command).
Retrieve	6.4.37	Retrieve MMS from network and store it on the smart-card.

multimedia message (MMS)		This command only exists on newer phones.
Submit multimedia message	6.4.38	Sends a MMS from the smart-card.
Display multimedia message	6.4.39	Displays a MMS stored on the smart-card on the phone's display.
Open channel	6.4.27	Open a new channel; this can be a TCP/UDP socket or other channel types.
Close channel	6.4.28	Closes a specified channel.
Received data	6.4.29	Receive data from a channel.
Send data	6.4.30	Send data on a channel.
Get channel status	6.4.31	Get channel status (open, closed etc.)
Service search	6.4.32	Lists available phone (terminal) services (typically protocol stacks)
Get service information	6.4.33	Get detailed information about a service. This information can be used with the OPEN CHANNEL command.
Declare service	6.4.34	Downloads smart-card side services into the phone (terminal). Note that a smart-card also can contain communications hardware (Bluetooth etc.) which the terminal can use.

Table 15: APDU commands for messaging and signaling.

Appendix E – List of smart card/SIM card manufacturers

In order to purchase developer SIM cards you have to contact the manufacturer to see if and where they have resellers in your country (these are typically consultant firms, payment solution providers and the like, unfortunately you might have to present them with a business case since they want to sell to you in volumes). Smart card readers are easy to obtain, just search for it on the Internet.

List of smart card resellers and manufacturers
- Tele Pak (http://www.tele-pak.com/plastic-cards/otacards.html), can provide samples.
- SMARTJAC (http://www.smartjac.se), Gemalto reseller based in Sweden.
- Gemalto (www.gemalto.com), cards and courses.
- Giesecke & Devrient Sm@rtCafé (www.gdm.de)
- Oberthur GalactiC (www.oberthurusa.com)
- Schlumberger Cyberflex (www.cardstore.slb.com)
- IBM Java Card (Peter Buhler at bup@zurich.ibm.com
- Aspects Software (www.aspects-sw.com)
- Microelectronica Espanola (www.exceldata.es)
- I'M Technologies (www.imcorporation.com)
- Datacard Aptura (www.datacard.com)
- Fujitsu HIPERSIM (www.fujitsu.com)
- Datakey Model 330J (www.datakey.com)
- WebKomputing (www.webkomputing.com)
- Philips Semiconductor (www.philips.com and www.nxp.com)

Appendix F – Example application source code

Source file JCExample.Java is listed (only one source file in the project). This source file plus the rest of the project's files can be found in the RAR archive (JavaCardExampleR99.rar). Source comes as-is with no warranty of any kind attached.

```
/*
 * JCExample
 * SIM Toolkit / Java Card example for Bachelors Thesis in Computer Engineering.
 * SIM card target type is R99.
 * Code was written by Peter Edsbäcker 2010, 2011.
 * Based on example skeleton code provided by Gemalto's Applet wizard.
 */
package jcexample; // Applet's package name

/*
 * Imported packages
 */
import sim.toolkit.*;
import sim.access.*;
import Javacard.framework.*;


public class JCExample
  extends Javacard.framework.Applet
  implements ToolkitInterface, ToolkitConstants
{
    private static final short MSG_MAINMENU_OFFSET = (short) 0;
    private static final short MSG_MAINMENU_LENGTH = (short) 9;

    private static final short MSG_NUMBER_TO_DIAL_OFFSET = (short) MSG_MAINMENU_LENGTH;
    private static final short MSG_NUMBER_TO_DIAL_LENGTH = (short) 14; // "Number to dial"

    private static final short MSG_NUMBER_TO_SMS_OFFSET = (short)
(MSG_NUMBER_TO_DIAL_OFFSET+MSG_NUMBER_TO_DIAL_LENGTH);
    private static final short MSG_NUMBER_TO_SMS_LENGTH = (short) 16; // "Number to SMS to"

    private static final short MSG_DIALING_OFFSET = (short)
(MSG_NUMBER_TO_SMS_OFFSET+MSG_NUMBER_TO_SMS_LENGTH);
    private static final short MSG_DIALING_LENGTH = (short) 7; // "Dialing"

    private static final short MSG_SENDING_SMS_OFFSET = (short)
(MSG_DIALING_OFFSET+MSG_DIALING_LENGTH);
    private static final short MSG_SENDING_SMS_LENGTH = (short) 11; // "Sending SMS"

    private static final short MSG_SUCCESS_OFFSET = (short)
(MSG_SENDING_SMS_OFFSET+MSG_SENDING_SMS_LENGTH);
    private static final short MSG_SUCCESS_LENGTH = (short) 7; // "Success"

    private static final short MSG_FAILURE_OFFSET = (short)
(MSG_SUCCESS_OFFSET+MSG_SUCCESS_LENGTH);
    private static final short MSG_FAILURE_LENGTH = (short) 7; // "Failure"

    private static final short MSG_MENU_OFFSET = (short)
(MSG_FAILURE_OFFSET+MSG_FAILURE_LENGTH);
    private static final short MSG_MENU_LENGTH = (short) 4; // "Menu"

    private static final short MSG_DIAL_NUMBER_OFFSET = (short)
(MSG_MENU_OFFSET+MSG_MENU_LENGTH);
    private static final short MSG_DIAL_NUMBER_LENGTH = (short) 13; // "1 Dial number"

    private static final short MSG_SEND_SMS_OFFSET = (short)
(MSG_DIAL_NUMBER_OFFSET+MSG_DIAL_NUMBER_LENGTH);
    private static final short MSG_SEND_SMS_LENGTH = (short) 10; // "2 Send SMS"

    private static final short MSG_EXIT_OFFSET = (short)
(MSG_SEND_SMS_OFFSET+MSG_SEND_SMS_LENGTH);
    private static final short MSG_EXIT_LENGTH = (short) 6; // "3 Exit"

    // SMSC dialing number protocol constant
         private static final byte SMS_TON_NPI = (byte) 0x081; // Local number, NPI
ISDN/telephone numbering plan.
         private static final byte VOICECALL_TON_NPI = (byte) 0x081; // Same here.
         private static final byte GI_DIGITSONLY = 0x008;
         private static final byte GI_HIDEINPUT  = 0x004;
    // Byte buffer containing the Applet's data strings
         // We pack everything together to save space. For a real application this might
         // be created by a pre-compiler or exist in a GSM file (for multiple languages)
         //
    private byte[] messageStrings =
    {
       // "JCExample"
       (byte)'J',(byte)'C',(byte)'E',(byte)'x',(byte)'a',(byte)'m',
(byte)'p',(byte)'l',(byte)'e',

       // "Number to dial" (14)
       (byte)'N',(byte)'u',(byte)'m',(byte)'b',(byte)'e',(byte)'r', (byte)' ',
       (byte)'t',(byte)'o',(byte)' ',(byte)'d',(byte)'i',(byte)'a', (byte)'l',

       // "Number to SMS to" (16)
       (byte)'N',(byte)'u',(byte)'m',(byte)'b',(byte)'e',(byte)'r', (byte)' ',
       (byte)'t',(byte)'o',(byte)' ',(byte)'S',(byte)'M',(byte)'S', (byte)' ',
       (byte)'t',(byte)'o',

       // "Dialing" (7)
       (byte)'D',(byte)'i',(byte)'a',(byte)'l',(byte)'i', (byte)'n',(byte)'g',

       // "Sending SMS" (11)
       (byte)'S',(byte)'e',(byte)'n',(byte)'d',(byte)'i',(byte)'n', (byte)'g', (byte)' ',
       (byte)'S',(byte)'M',(byte)'S',

       // "Success" (7)
       (byte)'S',(byte)'u',(byte)'c',(byte)'c',(byte)'e',(byte)'s', (byte)'s',

       // "Failure" (7)
       (byte)'F',(byte)'a',(byte)'i',(byte)'l',(byte)'u',(byte)'r', (byte)'e',

       // "Menu" (4)
       (byte)'M',(byte)'e',(byte)'n',(byte)'u',

       // "1 Dial number" (13)
       (byte)'1',(byte)' ',(byte)'D',(byte)'i',(byte)'a',(byte)'l', (byte)' ',
       (byte)'n',(byte)'u',(byte)'m',(byte)'b',(byte)'e',(byte)'r',

       // "2 Send SMS" (10)
       (byte)'2',(byte)' ',(byte)'S',(byte)'e',(byte)'n',(byte)'d', (byte)' ',
       (byte)'S',(byte)'M',(byte)'S',

       // "3 Exit" (6)
       (byte)'3',(byte)' ',(byte)'E',(byte)'x',(byte)'i',(byte)'t'
    };

    // Has to be in its own buffer for now.
    private byte[] MessageSendSMS =
    {
       // "Sending SMS"
       (byte)'S',(byte)'e',(byte)'n',(byte)'d',(byte)'i',(byte)'n', (byte)'g',
       (byte)' ', (byte)'S',(byte)'M',(byte)'S',
    };

    private byte[] MessageSMSBodyText =
    {
       // SIM card says hello!
       (byte)'S',(byte)'I',(byte)'M',(byte)' ',(byte)'C',(byte)'a', (byte)'r',
       (byte)'d',(byte)' ',(byte)'s',(byte)'a',(byte)'y',(byte)'s',(byte)' ',
       (byte)'h',(byte)'e',(byte)'l',(byte)'l',(byte)'o',(byte)'!'
    };


    private short[] MainMenu =
    {
         // First comes menu's title (ofs,len)
         MSG_MENU_OFFSET, MSG_MENU_LENGTH,

         // Then comes the items (ofs,len)
         MSG_DIAL_NUMBER_OFFSET, MSG_DIAL_NUMBER_LENGTH,
         MSG_SEND_SMS_OFFSET, MSG_SEND_SMS_LENGTH,
         MSG_EXIT_OFFSET, MSG_EXIT_LENGTH
    };



    // Mask defining the SIM Toolkit features required by the Applet
    // It is a bitmask with 1s reflecting the needed profiles.
    //
    private byte[] terminalProfileMask  =
{(byte)0x09,(byte)0x03,(byte)0x21,(byte)0x70,(byte)0x0D};

    // Volatile RAM temporary buffer for storing intermediate data and results
    // It is 180 bytes, enough for a long SMS + dialing number.
    //
    private byte[] tempBuffer;

    private boolean environmentOk = false;
    private boolean eventsRegistered;
    private byte menuEntryId_1; // SIM Root menu item id. Used for register/unregister

    /**
     * Constructor of the Applet
     * It is only executed during Applet installation.
     * All memory buffers should be created here (at least for the
     * JavaCard 2.x generation)
     */
    public JCExample()
    {
        // Create tempBuffer[] in RAM (to avoid EEPROM stress due to high update rates)
        tempBuffer = JCSystem.makeTransientByteArray((short)180, JCSystem.CLEAR_ON_RESET);

        // Register to the SIM Toolkit Framework
        ToolkitRegistry reg = ToolkitRegistry.getEntry();

        // Register the Applet under the EVENT_MENU_SELECTION event
        // PRO_CMD_SET_UP_CALL here means gurka
        //
        menuEntryId_1 = reg.initMenuEntry(
                 messageStrings, MSG_MAINMENU_OFFSET, MSG_MAINMENU_LENGTH,
            PRO_CMD_SET_UP_CALL,
            false, (byte)0, (short)0);

        // Register the other events usd by the Applet
        reg.setEvent(EVENT_CALL_CONTROL_BY_SIM);
        reg.setEvent(EVENT_PROFILE_DOWNLOAD);

        // Set the 'eventsRegistered' flag if there is no exception before it.
        eventsRegistered = true;
    }


    /**
     * Method called by the JCRE when the Applet is installed
     * The bArray contains installation parameters
     * These usable parameters are stored as LV-pairs (1 byte=Data length followed by
parameter data)
     * and follow in this order (data pairs starts at offset <bOffset>)
     * - Applet's instance AID
     * - Control information
     * - Applet data (the "command line" for the Applet instance)
     * If you want to fail installation please throw an exception with
     * ISO7816.throwit(exception_constant), see Java Card documentation.
     */
    public static void install(byte bArray[], short bOffset, byte bLength)
    {
        // Create the Applet instance
        JCExample JavaCardExample = new JCExample();

        // Register the Applet's instance with the JCRE.
        // Argument here is the AID parameter of the installation parameters (as seen above)
        //
        JavaCardExample.register(bArray, (short)(bOffset + 1), (byte)bArray[bOffset]);
    }



     /**
      * Method called by the SIM Toolkit Framework to trigger the Applet
      */
     public void processToolkit(byte event)
     {
        // Define the SIM Toolkit session handler variables
         // We don't fetch unused values directly here, everything takes extra time..
         //
        EnvelopeHandler          envHdlr;
        ProactiveHandler         proHdlr;
        ProactiveResponseHandler rspHdlr;
        EnvelopeResponseHandler  envRspHdlr;

        switch(event)
        {
        // The EVENT_PROFILE_DOWNLOAD happens after installation here you can check
        // if the SIM card and phones (ME) functionality is good enough for your
application.
        //
         case EVENT_PROFILE_DOWNLOAD:
          // Test that Mobile Equipment capabilities and card personalisation are compatible
          // with the Applet's requirements
          environmentOk = testAppletEnvironment();
          if (environmentOk) // ME capabilities and SIM are OK.
          {
            // Test if Applet events are registered and register if necessary
            if (!eventsRegistered)
              registerEvents(); // Applet can now respond to events and shows in root menu.
          }
          else
          {
            if (eventsRegistered)
               clearEvents(); // Applet no longer receives events and does not show in root
menu
           }
           break;

            // The EVENT_STATUS_COMMAND is used so Applet gets called
            // at specified intervals. We don't use it in this example.
            // case EVENT_STATUS_COMMAND:
            //    break;

            // Example of call control
            // This event is generated when the ME is about to dial a number.
            // The SIM can change dialling parameters (including phone number)
            // before dialling actually takes place. Here we just return "OK"
            //
            case EVENT_CALL_CONTROL_BY_SIM:
                 envRspHdlr = EnvelopeResponseHandler.getTheHandler();
                 envRspHdlr.postAsBERTLV((byte)0x9F, (byte)0x00); // (0x00 is a result
code)
                 break;

            case EVENT_EVENT_DOWNLOAD_CALL_CONNECTED:
                 // Call connected. Here you could for example send DTMF tones
                 // with command PRO_CMD_SEND_DTMF (ETSI section 11.11)
                 // It is NOT typically supported to display messages etc here on most
phones!!
                 break;

            // EVENT_MENU_SELECTION : Root menu selection done
            //
            case EVENT_MENU_SELECTION:
                // Get the references of the required SIM Toolkit session handlers
                proHdlr = ProactiveHandler.getTheHandler();
                envHdlr = EnvelopeHandler.getTheHandler();

                // Get the identifier of the SIM Tooklit menu item selected by the user
                byte itemId = envHdlr.getItemIdentifier();

                // If selected item identifier matches registered menu item identifier ...
                if (itemId == menuEntryId_1)
                {
                 RootMenuHandler();
                }
                break;
        }
    }


    /**
     * Method called by the JCRE, once selected
     */
     public void process(APDU apdu)
     {
          // Method not implemented - this Applet only processes SIM Toolkit events
     }


     public void RootMenuHandler()
     {
         while(true)
         {
           byte item = displayMenu(messageStrings, MainMenu, (byte)0);
           if (item<0 || item >= 2) // Error or "exit" selected?
                 break;

           switch (item)
           {
           case 0: // Dial number.
                 AskForInputAndDialNumber();
                 break;

           case 1: // Send SMS
                 AskForInputAndSendTestSMS();
                 break;
           }
         }
     }


     private void AskForInputAndDialNumber()
     {
         short len = getInput(messageStrings, MSG_NUMBER_TO_DIAL_OFFSET,
        MSG_NUMBER_TO_DIAL_LENGTH,
           tempBuffer, (short)0, // Result into tempBuffer at offset 0.
           (short)1, (short)16, GI_DIGITSONLY); // Min 1 digit, Max 16 digits.

        if (len > 0) // Valid user respose.
        {
           // Dial number in tempBuffer, connect, wait until disconnect.
           // Note that packed data is placed in workBuffer after offset len
           //
           boolean success = dialNumber((short)0, len);
           DisplayOperationResult(success);
        }
     }


     private void AskForInputAndSendTestSMS()
     {
        short len = getInput(messageStrings, MSG_NUMBER_TO_SMS_OFFSET,
        MSG_NUMBER_TO_SMS_LENGTH,
           tempBuffer, (short)0, // Result into tempBuffer at offset 0.
           (short)1, (short)16, GI_DIGITSONLY); // Min 1 digit, Max 16 digits.

        if (len > 0) // Valid user respose.
        {
          short rc = sendSMS(
            true, // Send as ASCII 7-bit (standard 160 character limit).
            MessageSMSBodyText, (short)0, (short)MessageSMSBodyText.length, // Text to send
to target.
            tempBuffer, // tempBuffer contains SMS dialling number.
            (short) 0,  // tempBuffer offset for SMS number.
            len,        // tempBuffer number length
            MessageSendSMS, // Message to display while dispatching SMS (from ofs 0,
length=.length)
            tempBuffer,
            len); // Transient data buffer, at least 160 characters after ofs "len" (16+160)

            DisplayOperationResult(rc>=0);
        }
     }

     /**
      * Displays "success" or "failure" to the user.
      * @param displaySuccess
      */
     private void DisplayOperationResult(boolean displaySuccess)
     {
          if (displaySuccess)
            displayMsgWaitRsp(messageStrings, MSG_SUCCESS_OFFSET,     MSG_SUCCESS_LENGTH);
          else
            displayMsgWaitRsp(messageStrings, MSG_FAILURE_OFFSET,
MSG_FAILURE_LENGTH);
     }


    /**
     * Tests whether :
     * - the Mobile Equipment supports the SIM Toolkit functionalities required by the
Applet.
     * - the card's file system contains the files required by the Applet.
     *
     * Returns :
     * - boolean result : true if the ME and card comply with the Applet's requirements,
     *                    false otherwise.
     */
    private boolean testAppletEnvironment()
    {
        // Check that the ME (phone) supports the toolkit features required by the Applet.

        // The .check method with one argument can be used to see if the ME (phone) support
certain command(s)
        // (For example the PRO_CMD_xxx constants)
         // Here we use a bit-mask array to check multiple features at once,
         // we expect features with respective bit set to "1" to exist.
         // Bit-0 in the array refers to existence of function 0 and so on.
         //
        return (MEProfile.check(
         terminalProfileMask, (short)0, (short)terminalProfileMask.length)
         );
    }


    /**
     * Registers the events used by the Applet (excepting EVENT_PROFILE_DOWNLOAD which is
     * not cleared) and sets the eventsRegistered flag to 'true'.
     */
    private void registerEvents ()
    {
        ToolkitRegistry reg = ToolkitRegistry.getEntry();

        // Enable EVENT_MENU_SELECTION for menuEntryId_1
        reg.enableMenuEntry(menuEntryId_1);

        // Register EVENT_STATUS_COMMAND every 30 sec (not used)
        // reg.requestPollInterval((short)30);

        // Register all the other events we listen to in the Applet.
        //
        reg.setEvent(EVENT_CALL_CONTROL_BY_SIM); // When ME is about to dial.
        // reg.setEvent(EVENT_EVENT_DOWNLOAD_CALL_CONNECTED); // When ME has connected.
        // reg.setEvent(EVENT_EVENT_DOWNLOAD_CALL_DISCONNECTED); // When ME has
disconnected.

        // Set the eventsRegistered flag (no exception happened before it)
        eventsRegistered = true;
    }

    /**
     * Clears the events used by the Applet (excepting EVENT_PROFILE_DOWNLOAD so that
     * the Applet can continue testing its environment and register the events if a
     * compliant environment is detected).
     * Also sets the eventsRegistered flag to 'false'.
     */
    private void clearEvents ()
    {
        ToolkitRegistry reg = ToolkitRegistry.getEntry();

        // Disable EVENT_MENU_SELECTION for menuEntryId_1
        reg.disableMenuEntry(menuEntryId_1);

        // Clear EVENT_STATUS_COMMAND (Not used)
        // reg.requestPollInterval(POLL_NO_DURATION);

        // Clear all the other events used by the Applet
        reg.clearEvent(EVENT_CALL_CONTROL_BY_SIM);

        // Call progress monitoring not implemented in this example:
        // reg.clearEvent(EVENT_EVENT_DOWNLOAD_CALL_CONNECTED);
        // reg.clearEvent(EVENT_EVENT_DOWNLOAD_CALL_DISCONNECTED);

        // Set the eventsRegistered to false
        eventsRegistered = false;
    }


    /**
     *
     * @param send_7bit : Send ASCII 7-bit (standard 160 character limit). Otherwise 8-bit
(140 char limit)
     * @param databuff : Buffer to send
     * @param databuff_ofs : Offset in send buffer
     * @param databuff_len : Length of data to send
     * @param smsc_address_data : SMSC address buffer, digit as byte value 0-9 (SMS number
destination)
     * @param smsc_address_ofs : Offset in buffer
     * @param smsc_address_len : Length of SMSC number.
     * @param sending_user_message : Text to display to the user while sending
     * @param work_buffer : Transient work buffer, at least 160
bytes+4+smsc_address_length/2
     * @return
     */
         public short sendSMS(
                 boolean send_7bit, // Send ASCII 7-bit (standard 160 character limit).
Otherwise 8-bit (140 char limit)
                 byte[] sms,
                 short smsOffset,
                 short smsLength,
                 byte[] smsc_address_data,
                 short smsc_address_ofs,
                 short smsc_address_len,
                 byte[] userMessageDuringSending, // Message to display while dispatching
SMS (from ofs 0, length=.length)
                 byte[] workBuffer,
                 short workBufferOffset) // Transient data buffer, at least 160 characters+
         {
                 ProactiveHandler proHdlr = ProactiveHandler.getTheHandler();
                 short rc;
                 short pos = workBufferOffset;
                 byte tpdu_dcs;

                 final byte CMD_QUALIFIER_PACKING_NOT_REQUIRED = (byte) 0;

                 // Now Construct the SMS TPDU in the recordData
                 // see GSM 03.40 for details
                 workBuffer[pos] = (byte) 0x01; // SMS-Submit Type
                 workBuffer[(short)(pos+1)] = (byte) 0x01; // Message reference
                 pos+= 2;

                 // TP-Destination Address
                 //
                 short adn_length = buildADNumber(
                     true,
                     workBuffer,
                     pos, // Add encoded address at offset 2.
                          smsc_address_data, smsc_address_ofs, smsc_address_len );

                 pos+= adn_length;
                 workBuffer[pos++] = (byte) 0x00; // TP-PID 0x00 (Protocol identifier)

                 tpdu_dcs = (byte)0x00;
                 if ( !send_7bit )
                         tpdu_dcs = (byte) 0x04; // 8bit data

                 // DCS
                 // Bit5  : 1=compressed                                         [0x020]
                 // Bit4  : 1=Use bit1,0 as message class, 0=bit1,0 is reserved [0x010]
                 // Bit3,2: Alphabet (0=GSM 7bit,1=8bit,2=UCS2,3=reserved)
[0x008,0x004]
                 // Bit1,0: 0=Class0, 1=MobilePhoneSpecific, 2=SIM specific, 3=TE specific
                 //  Class0 is "Flash SMS" and are not normally stored by the phone.
                 //
                 workBuffer[pos++] = tpdu_dcs; // TP-DCS 0x00 (Data coding scheme)
                 workBuffer[pos++] = (byte) smsLength; // (byte) (smstpduLength -
start_offset);

                 // Pack data to 7 bit before sending
                 if ( send_7bit )
                 {
                         short acc = 0;
                         short bit_index=0;

                         for(short si=0; si<smsLength; si++)
                         {
                                 acc|= ((sms[(short)(si+smsOffset)]&0x0ff) << bit_index);
// Highest allowed is 7
                                 bit_index+= 7;
                                 if ( bit_index >= 8 )
```

```
                                {
                                        workBuffer[pos++] = (byte) acc;
                                        acc>>>= 8; // Unsigned shift right high. Do NOT 
allow high sign bit to spread!
                                        bit_index&= 0x07; // Mask bit index to get new 
current index.
                                }
                        }

                        if ( bit_index != 0 )
                        {
                                if ( bit_index == 1 ) // Can exactly fit one extra byte, 
it has to be a space..
                                        acc = (short)((short)' ' << bit_index);
                                workBuffer[pos++] = (byte) acc;
                        }
                }
                else
                {
                        Util.arrayCopyNonAtomic(sms, (short)smsOffset, workBuffer, pos, 
smsLength);
                        pos+= smsLength;
                }

                try
                {
                        // Sending the message, command 19 = 0x013

        proHdlr.init(PRO_CMD_SEND_SHORT_MESSAGE,CMD_QUALIFIER_PACKING_NOT_REQUIRED, 
DEV_ID_NETWORK);
                        if ( userMessageDuringSending != null )
                                proHdlr.appendTLV((byte)
(TAG_ALPHA_IDENTIFIER),userMessageDuringSending, (byte)0, (short)
userMessageDuringSending.length);

                        proHdlr.appendTLV((byte) (TAG_SMS_TPDU),
                                workBuffer, workBufferOffset, (short)(pos-
workBufferOffset));

                        rc = proHdlr.send();
                }
                catch(Exception ex)
                {
                        rc = (int) -256;
                }

                return rc;
        }


        /**
         * Dials given number.
         * Places packed temporary data in tempBuffer after
         * position numberTempBufferOffset+numberLength.
         *
         * Before dialing the ME creates a EVENT_CALL_CONTROL_BY_SIM event
         * where we can adjust dialing parameters before actual dial is done.
         * Typically the call ends up with these events:
         * EVENT_EVENT_DOWNLOAD_CALL_CONNECTED
         * followed by a
         * EVENT_EVENT_DOWNLOAD_CALL_DISCONNECTED
         */
        public boolean dialNumber(short numberTempBufferOffset, short numberLength)
        {
                ProactiveHandler proHdlr = ProactiveHandler.getTheHandler();

                // Put the packed BCD data into "tempBuffer" after given input string.
                short packed_offset = (short)(numberTempBufferOffset + numberLength);

                short packed_len = buildADNumber(false, tempBuffer, packed_offset,
                        tempBuffer, numberTempBufferOffset, numberLength);

                // See GSM 11.14 spec, "SET UP CALL" section 6.4.13 and 6.6.12
                // The init command creates a BER-TLV with a "command details" tag
                // added automatically (see section 11.6). Here 0x00 means "call only if
                // no other call is in progress". DEV_ID_NETWORK is the command qualifier 
byte.
                //
        proHdlr.init(PRO_CMD_SET_UP_CALL, (byte)0x00, DEV_ID_NETWORK);
        proHdlr.appendTLV(TAG_ALPHA_IDENTIFIER, messageStrings, MSG_DIALING_OFFSET,
MSG_DIALING_LENGTH);
        proHdlr.appendTLV(TAG_ADDRESS, tempBuffer, packed_offset, packed_len); // TON/NPI
byte followed by packed BCD number
        proHdlr.send();

                ProactiveResponseHandler prh = ProactiveResponseHandler.getTheHandler();
                return ( prh.getGeneralResult() == 0 );
        }



        /**
         * buildADNumber
         * Creates a SMS/Dialling number from the input digits (bytes 0-9)
         * The output is in a packed BCD format (4 bits per digit)
         * Note: A NULL byte(0) or space ends the smsc_number, even if its array length is
longer.
         * Returns: Length of generated data.
         */
        private static short buildADNumber(
                boolean buildSmscNumber, // If false it builds a phone number.
                byte[] dstBuffer,
                short dstBufferOffset,
                byte[] number,        // 0-9 as ASCII BYTES.
                short numberOffset,
                short numberLength)
        {
                short inputDigits;

                for(inputDigits=0; inputDigits<numberLength; inputDigits++)
                {
                        if ( number[(short)(numberOffset+inputDigits)] <= 32 )
                                break; // Also break off if we see a space or 0
                }

                // Calculate length of BCD number in bytes.
                short result_length = (short)((short)(inputDigits+1) >> 1);

                if (buildSmscNumber)
                {
                        result_length+= 2; // Two info header bytes.
                        if (dstBuffer == null) // Just calculating length?
                                return result_length;

                        // 2 hdr bytes plus 4 bits per digit, padded with 0x0f if
necessary (round size up)
                        // See ETSI 102 / 8.1 Address, page 116.
                        dstBuffer[dstBufferOffset++] = (byte) inputDigits;
                        dstBuffer[dstBufferOffset++] = SMS_TON_NPI;
                }
                else
                {
                        result_length++; // One header byte.
                        if (dstBuffer == null) // Calculating length?
                                return result_length;

                    // For information about EF-ADN packed numbers see the GSM 11.11
document, section 11.1
                        //
                        dstBuffer[dstBufferOffset++] = VOICECALL_TON_NPI;
                }

                // Packed as 4 bit BCD, padded with 0x0f0 if needed (if unaligned
data/uneven length)
                //
                short number_exit = (short)(numberOffset + inputDigits);

                while (numberOffset < number_exit)
                {
                        byte data = (byte) (number[numberOffset++] - '0');
                        dstBuffer[dstBufferOffset] = data;
                        if (numberOffset >= number_exit) // Odd length and last digit!
                        {
                                dstBuffer[dstBufferOffset]|= 0x0f0; // Pad with BCD 0x0f

                                break;
                        }
                        dstBuffer[dstBufferOffset++]|= (byte) ( (number[numberOffset++] -
'0') << 4 );
```

```
                }

                return result_length;
        }

        /**
         * Displays given message on the ME a few seconds (default time, see ETSI document)
         * @param msg : Buffer (8-bit GSM character set)
         * @param msgOffset : Offset of message in buffer
         * @param msgLength : Length of data
         */
        private void displayMessage(byte[] msg, short msgOffset, short msgLength)
        {
                ProactiveHandler proHdlr = ProactiveHandler.getTheHandler();
                proHdlr.initDisplayText( (byte)0x00, DCS_8_BIT_DATA, msg, msgOffset,
msgLength);
                proHdlr.send();
        }


        /* displayMsgWaitRsp (Command 21 decimal = 0x015)
         * RETURNS:
         * <0 : Error
         *  0 : User pressed NO
         *  1 : User pressed YES (OK etc)
         *
         * displayMessage is explained in (R5) TS 102 223 section 6.6.1
         * IMPORTANT: The QUALIFIER bits are described under 8.6!!!
         * DISPLAY TEXT:
         *  bit 0: 0 = normal priority ; 1 = high priority.
         *  bits 1 to 6: = Reserved
         *  bit 7: 0 = clear message after a delay, 1 = wait for user to clear message.
         *
         * Duration of displayed message is explained in 8.8:
         *       Byte(s) Description      Length
         *        0      Duration tag     1
         *    1  Length = '02'    1
         *        2 Time unit             1       -       '00' minutes , '01' seconds,
'02' tenths of seconds, All other values are reserved.
         *        3      Time interval    1 - in units (0=reserved)
         *
         */
        private short displayMsgWaitRsp(byte[] msg, short pos, short len )
        {
                short rc;

                ProactiveHandler proHdlr = ProactiveHandler.getTheHandler();
                proHdlr.initDisplayText((byte) 0x080, DCS_8_BIT_DATA, msg, pos, len );

                // FYI: This is how to do the same in "raw" mode
                // proHdlr.init( (byte) PRO_CMD_DISPLAY_TEXT, (byte) 0x81,(byte)
DEV_ID_DISPLAY);
                // proHdlr.appendTLV( (byte) (TAG_TEXT_STRING), (byte) 0x04, msg, pos, len
);

                if ( proHdlr.send() != 0 )
                        return -2; // Command error.

                ProactiveResponseHandler prh = ProactiveResponseHandler.getTheHandler();

                rc = 0; // Responded NO (or some fishy key)
                if ( prh.getGeneralResult() == 0 )
                        rc = 1; // Responded YES

                return rc; // Responded NO
        }



        /**
         * Get input from user.
         * @param menu_text
         * @param out_response
         * @param out_response_offset
         * @param minResponse
         * @param maxResponse
         * @param qualifier : The GI_xxx constants defined in the header.
         * @return
         */
        private short getInput(
			byte menuText[],
			short menuTextOffset,
			short menuTextLength,
			byte out_response[],
			short out_response_offset,
			short minResponse,
	short maxResponse,
	byte qualifier)
	{
		ProactiveHandler ph = ProactiveHandler.getTheHandler();

		// qualifier, ...
		ph.initGetInput(
			(byte) (qualifier & GI_HIDEINPUT),
			DCS_8_BIT_DATA,
			menuText, menuTextOffset, menuTextLength,
			minResponse,
			maxResponse);

		if ( ph.send() != 0 )
			return -1;

		ProactiveResponseHandler prh = ProactiveResponseHandler.getTheHandler();

		short out_responseLength = (byte)prh.getTextStringLength();
		prh.copyTextString(out_response, out_response_offset);

		if ( (qualifier & GI_DIGITSONLY)!=0 )
		{
			for(short i=0; i<out_responseLength; i++)
			{
				short bin_digit = (short) (
(out_response[(short)(i+out_response_offset)]-'0') & 0x0ff );

				if ( bin_digit > 9 )
					return -2; // Invalid digits.
			}
		}
		return (out_responseLength);
	}



	/**
	 * Defines a menu.
	 * Data is tuples of (offset,length)
	 * Tuple 0 is title
	 * Tuple 1..N are the menu items, which gets menu index 0..N-1
	 * @param menuDataBuffer
	 * @param menuDefinition
	 * @param selectedItemIndex if >=0, preselects given menu index.
	 * @return
	 */
	public byte displayMenu(
		byte[] menuDataBuffer,
		short[] menuDefinition,
		byte   selectedItemIndex)
	{
		short menu_ofs, menu_exit, item_start_ofs, item_length;
		ProactiveHandler proHdlr = ProactiveHandler.getTheHandler();

		// ---| PRE MENU SETUP
		proHdlr.init(PRO_CMD_SELECT_ITEM, (byte)0x080, DEV_ID_ME);

		// Menu header
		proHdlr.appendTLV(TAG_ALPHA_IDENTIFIER, menuDataBuffer, menuDefinition[0],
menuDefinition[1]);

		short menu_data_offset = (short) 2;
		byte item_index = (byte) 0; // Added item's index. 0 = first item etc.

		// Menu items
		while (menu_data_offset < menuDefinition.length)
		{
			proHdlr.appendTLV((byte)(TAG_ITEM | TAG_SET_CR), item_index,
				menuDataBuffer, // Item text buffer.
				menuDefinition[menu_data_offset], // Offset
				menuDefinition[(short)(menu_data_offset+1)]); // Length

			item_index++;
```

```
                        menu_data_offset+= 2;
                }

                // pre-selected item (where the "cursor" is placed)
                if (selectedItemIndex>=0) // If we have a given start index >=0
                        proHdlr.appendTLV( TAG_ITEM_IDENTIFIER, selectedItemIndex);

                if ((selectedItemIndex = proHdlr.send()) < 0)
                        return selectedItemIndex; // Negative error codes.

                ProactiveResponseHandler prh = ProactiveResponseHandler.getTheHandler();
                if ( prh.getGeneralResult() != 0 )
                        return -2; // Return -2 if general result fails.

                return (byte) prh.getItemIdentifier(); // becomes index 0..N-1
        }

}
```