Master Thesis

Design and Implementation of a Java Card Operating System for Design Space Exploration on Different Platforms

Michael Lafer, BSc

Institute for Technical Informatics
Graz University of Technology
Head of the Institute: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Assessor:	Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
Advisor:	Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
	Dipl.-Ing. Reinhard Berlach

Graz, Januar 2014

Kurzfassung

Chipkarten (Smart Cards) haben sich zu einem ständigem Begleiter von vielen Menschen entwickelt und ermöglichen ihren Besitzern Leistungen der Krankenversicherung in Anspruch zu nehmen (ecard, Österreich), Finanztransaktionen zu tätigen (EMV) oder mobil zu telefonieren (SIM-Karte).

Um im heterogenen Umfeld von Chipkarten-Platformen die Abhängigkeit von einer bestimmten Architektur zu verringern wurde *Java Card* als Abstraktionsebene eingeführt und stellt eine Virtuelle-Maschine, samt mehrerer Bibliotheks-Klassen, zur Verfügen, wodurch Anwendungen (Applets) platformunabhängige realisiert werden können. Durch die hohen Anforderungen bezüglich Sicherheit und Leistung (z.b. Energieaufnahme) sind Chipkarten einer ständigen Weiterentwicklung unterworfen. Hierdurch ergibt sich des Weiteren die Notwendigkeit bestehende *Java Card* Implementierungen laufend anzupassen und zu erweitern.

Diese Masterarbeit beschäftigt sich mit dem Entwurf und der Implementierung eines *Java Card* Betriebssystems (JCOS) welches kompatibel zu bestehenden Implementierungen ist und zugleich leicht portiert und erweitert werden kann. Im Gegensatz zu einer kommerziellen Implementierung, steht nicht die Optimierung für eine spezielle Plattform im Vordergrund, sondern die universelle Einsetzbarkeit und eine gute Modifizierbarkeit zum Zwecke der Design-Space-Exploration.

Abstract

Smart cards have become a constant companion of many people. They are used to receive health insurance benefits (ecard, Austria), to make financial transactions (EMV) or to log into a cellular network (SIM).

Smart cards utilize a very heterogenous set of platform architectures which make it difficult to port applications. To overcome this problem, *Javacard* was introduced, working as an abstraction layer. Applications are provided with a virtual machine and a set of mandatory library classes. Due to high demands in the field of security and performance (especially power effectiveness) smart cards undergo a permanent advancement. Every one of this new or enhanced platforms needs an adapted *Javacard* implementation.

This master thesis deals with the design and implementation of a standard conform *Javacard* implementation which is easily to port and to maintain. In contrast to existing, usually commercial, implementations, we don't focused on optimizing for a certain platform, instead we strived for versatility and expandability in order to use the outcome of this thesis for further design space exploration.

Keywords

smart cards, hardware abstraction, virtual machines, javacard, portability, maintainability

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....	....
date	(signature)

Acknowledgment

This master thesis was carried out on the Institute for Technical Informatics at the Technical University of Graz during the year 2013/2014.

I want to thank all people at Institute for Technical Informatics who supported me during working on my master thesis. Especially I want to thank my two supervisors Prof. Dipl.-Ing. Dr.techn. Christian Steger and Dipl.-Ing. Reinhard Berlach.

I also want to thank my parents who supported me throughout my student days.

Graz, January 2014 Michael Lafer

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The long announced era of Ubiquitous Computing has finally arrived. A big part of this development can be attributed to smart cards. They accompany us as credit cards[1], health insurance cards [2]or inside our mobile phones [3]. Thus every one of us usually always carries a least one smart card.

This massive distribution of smart cards has also drawn a lot of interest into research and development of a new generations of smart cards. Also at the Graz University of Technology some projects engage in research on smart cards, one of them is the CoCoon project (see Section 1.1). Our thesis will supply them with a software platform for Java Card to allow faster and better testing of new smart card features.

1.1 The CoCoon Project

The CoCoon Project deals with codesigned countermeasures against malicious Java Card applications. [1]

The final goal is to enable Issuer Centric Ownership of smart cards. This allows the customers to carry only a single smart card with them which can run different application e.g. for banking or health insurance card. The problem is, that the user can download arbitrary Java Card application also from untrusted sources to these smart card. The untrusted application must not be allowed to interfere with other (possible critical eg. for financial transactions) applications.

Another aspect is to protect applications from the insecure environment a smart card is placed in. As a smart card can get lost or can be stolen, it is quiet possible that an attacker has physical access to the smart card.

[1]EMV standard for financial transactions `http://www.emvco.com/`
[2]ecard, health insurrance card in Austria `http://www.chipkarte.at/`
[3]Called subscriber identity module for registration to cellular network.

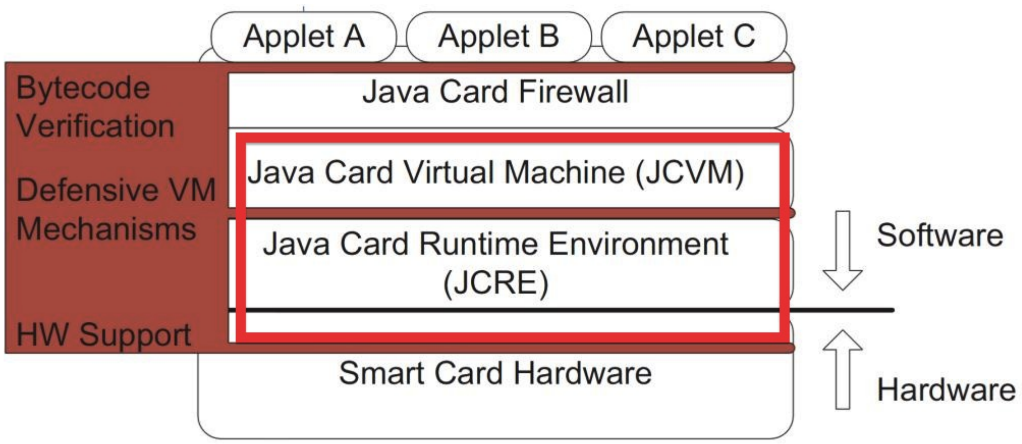


Figure 1.1: This figure shows the abstraction layers of the Java Card implementation used in the CoCoon project. Our work will involve the hardware layer as we have to support different hardware platforms and the VM respective the RE. The firewall and specific applets are not of interest to us. [1]

As our thesis deal with implementing a Java Card RE and VM for testing purpose we will operate in the middle layers of the Java Card stack (see figure Figure 1.1).

1.2 Motivation

As mentioned in the past section the CoCoon project explores new codesigned features for smart cards. This implies a Hardware (HW) and a Software (SW) part. The existing HW is defined by an ISA, a memory architecture and additional peripheries (see Section 2.3). For the SW part only the application interface is well defined (the Java Card standard). The implementation of the interface functionality differs for every card supplier. As smart cards have very spare resources, these implementation are usually highly optimized to specific platform used by the card supplier. This is very useful for smart cards in production environment, as it allows faster transactions, but not in a development environment where SW feature should be tested across different platforms. Another problem is that nearly all implementation are closed source under a proprietary license which also doesn't aid academic research and development.

The goal of our project is to develop a Java Card implementation for academic research which overcomes this problems. It should be designed to aid design space exploration for both SW and HW features. To enable this, we it will place a lot of attention on maintainability (to easily integrate new SW features) and (portability to easily test new HW features). As the source code is developed during a master thesis it will be freely available for research and will avoid license problems.

1.3 Outline

Our work is partitioned in the following chapters.

Chapter 2 will give an outline other foundations and existing approaches. In Section 2.1 we will deal with developing platform independent SW and how to best abstract HW features. In Section 2.2 we will give an overview of VM technologies and existing approaches of bringing VMs to embedded systems. In Section 2.3 we will focus on existing smart cards and specially on the Java Card technology.

Chapter 3 will show premises and ideas for solving the stated problems. In Section 3.1 we will discuss the functional and nonfunctional requirements for our implementation. In Section 3.2, Section 3.3 and Section 3.4 we will show the proposed system architecture for our implementation, including a separation in layers and modules. In Section 3.5 we will discuss which development tools we decided to use an why the think this tools will aid us in meeting our design goals.

Chapter 4 will show up with specific solution for the requirements stated in Section 3.1 and based on our design decisions in Section 3.2 and Section 3.5.

Chapter 5 will show how good our implementation (described in Chapter 4) meets the requirements stated in Section 3.1.

Chapter 6 will give a conclusion of our work and will show an outlook to possible future development.

Chapter 2

Related Work

This chapter will deal with some foundational topics of our work. First, in Section 2.1, we will deal with hardware abstraction concepts and we will look at existing solutions in different fields. Subsequently, in Section 2.2, we will give an introductions to Virtual Machine (VM)s and will present different approaches and implementations. At last, in Section 2.3, we will explain the Java Card platform, including some foundations of smart cards and an insight to the development process.

2.1 Hardware Abstraction

This section will focus on the attempt to hide the complexity of modern HW from application SW (and subsequently the SW developer). We will give a short introduction to general concepts of hardware abstraction and take a closer look to some real world implementations.

2.1.1 Concepts

Definition

The Hardware Abstraction Layer (HAL) can be defined as SW that directly depends on the hardware. Examples for this are the code to boot a system or code for configuration and access to HW resources. This piece of SW is used to hide details of the architecture from other SW parts (e.g. the Operating System (OS)). Device drivers are a related concept, as they also abstract HW resources (usually I/O). The hardware dependent part can be seen as part of the HAL. [2]

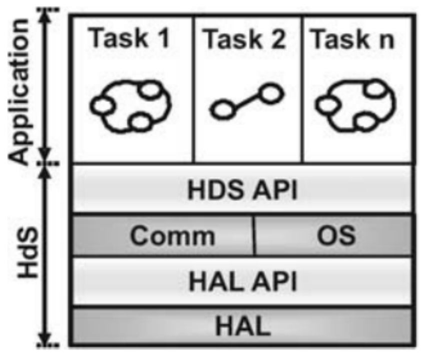


Figure 2.1: Example of a SW stack using a HAL. HAL, Comm and OS form the HW dependent SW layer, code for the application layer can be HW independent. [3, Page 71]

Basic Techniques

A general example for the organization of SW using a HAL can be seen in Figure 2.1. The lowest layer is the HAL, which incorporates processor specific code (e.g. code to boot a system, to handle interrupts or task switching) and SW routines to access present HW resources. It depends completely on the underlying HW. The HAL provides an interface (*HAL API*) to the next layer consisting of two SW modules. One module (*Comm*) handles communication with subsystems and HW components. The other module (*OS*) supplies SW routines for task scheduling, inter task communication and interrupt management. These two modules provide an interface (*HDS API*) to tasks which can be now written in HW independent code. The *HAL*, the *Comm* and the *OS* form the HW dependent SW layer. On top of this layer resides the application layer which can consists of multiple tasks. [3, Chapter 4]

Advantages of Hardware Abstraction

As mentioned before, when using anHAL with a well defined interface, applications which uses this interface can be developed independently of the underlying HW. This makes it easy to port this application to a different platform as only a HAL implementing the same interface must be provided.

Beside the reuse of SW, another advantage of using a HAL is to allow concurrent design of HW and SW (see Figure 2.2). Usually the SW development can not start before the HW design is finished, which will take a lot of time because of a pure sequential design process. By defining a clear interface between the two parts, both design processes can be done at the same time. [3, Chapter 4]

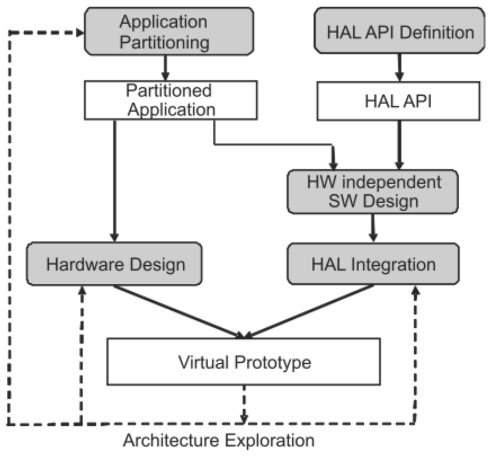


Figure 2.2: This figure shows the advantages of using a HAL for the development process. After application portioning and HAL interface definition the design for HW and SW can be done at the same time. [3, Page 69]

2.1.2 Hardware Abstraction on the PC platform

One main task of a all common desktop OS is the abstraction of HW resources. This means a reduction in complexity of the provided interfaces, for example presenting a storage devices as a hierarchical file system. Thus application SW can access data through simple open/close and read/write commands instead of dealing with mechanical and electronic properties of the device, like positioning the disk arm of a disk. This reduction in complexity of the HW interfaces (some also call it beautification, see Figure 2.3) also allows to exchange HW parts, without having to modify higher SW layers, as they only use the abstract interface. [4, Chapter 1]

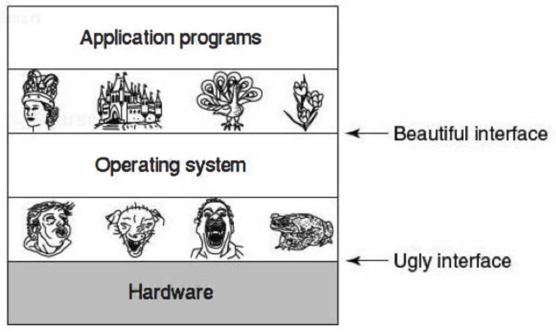


Figure 2.3: This figure shows how Tanenbaum sees hardware abstraction. *Ugly* interfaces of HW resources are turned into *beautiful* OS interfaces. [4, Page 5]

Because of this premises, Personal Computer (PC) OS make wide use of HW abstraction. For nearly all HW resources some functionality is hidden from application SW. Examples for partial abstraction are the CPU where most of the SW runs in a restricted mode and cannot execute all commands. The memory is seen applications as a linear virtual memory which can be distributed anywhere in the physical memory. An even higher level of abstraction is present for most other HW resources like I/O devices. As mentioned before mass storage is accessed through a filesystem instead of device specific commands. This abstraction is done by two components of the PC OS, the kernel and a set of device drivers. For both components various approaches exists which we will discuss in more detail in the following sections. [4, Chapter 2 to 5]

OS Kernel

The kernel is responsible for the earlier mentioned partial abstraction of CPU and memory. Thus it depends on the underlying HW and hides HW functionality and complexity from other SW parts.

On all common PC OS applications are grouped into processes. Each process can utilize a virtual CPU virtually all the time and multiple process can run virtually simultaneous. This paradox is solved with multiprogramming which means the scheduler selects a process, allows it to run for a certain time, safes its state to an external memory and selects and runs another process. Because of the fast switch between multiple processes they seem to run parallel and consistently. The process itself doesn't have to care about this switches and doesn't have to know how it is supported by the physical CPU. This functionality is hidden by the OS kernel and only an abstract API for settings like priorities is offered to the processes. [4, Chapter 2]

Processes cannot access memory directly, instead they always see a virtual address space of size $2^{32} - 1$ (or $2^{64} - 1$ depending on the used OS) regardless of the amount of available physical memory. The virtual address space is organized in virtual pages which are mapped to page frames in the physical memory by the Memory Management Unit (MMU). Page frames are only reserved in the physical memory when a virtual page is used. The page frames can be placed anywhere in the physical memory or can even be loaded and unloaded dynamically by the kernel (e.g. saved to mass storage). Also in this case, the process doesn't have to care about it. The MMU is operated solely by the kernel. [4, Chapter 3]

Device Driver

On a common PC OS, application access external devices, like mass storage or I/O devices, via device drivers. The set of all possible devices can be roughly separated in block and character devices (network drivers are somewhere in between, see next paragraph). Both are expected to support basic function, like read/write blocks and seek operation on the first one and send/receive characters on the latter one. To facilitate the interchange of device drivers, devices are further grouped into categories of similar types, like hard disks, keyboards etc. This grouping allows the definition of a larger set of mandatory functions, so a standard driver interface can be defined. Thus all SW expect the device driver can be developed device independently and also some code used for the device driver can be device independent (e.g. routines for buffering data). [4, Chapter 5]

A survey of the Linux kernel (Version 2.6.37.6, dated April 2011) found 3,217 distinct device driver. A lot of them support multiple devices, as this pool of device driver supports over 14,000 devices. Linux divides driver in three main categories which have additional subcategories: character, block and network. They also found that a significant part of device driver does not only message parsing between kernel and the device but also does computation like checksum calculation. Many Linux drivers don't interact directly with HW devices, instead they use a bus (PCI, USB, etc.). This gives another opportunity for hardware abstraction as these buses usually have a standardized interface. Device driver can be easier isolated and used on another platform which has a similar bus interface. [5]

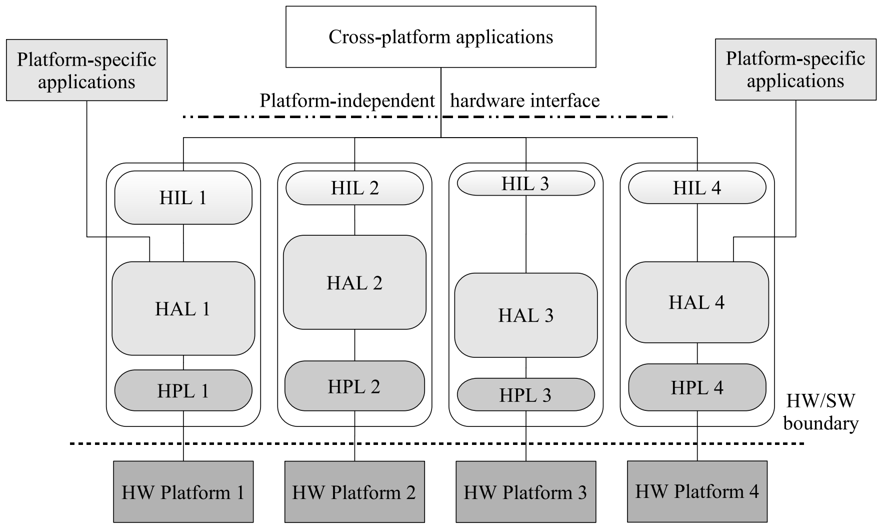

Cross-platform applications
Platform-specific applications
Platform-specific applications
Platform-independent hardware interface
HIL 1
HIL 2
HIL 3
HIL 4
HAL 1
HAL 2
HAL 3
HAL 4
HPL 1
HPL 2
HPL 3
HPL 4
HW/SW boundary
HW Platform 1
HW Platform 2
HW Platform 3
HW Platform 4

Figure 2.4: Hardware abstraction using a three layered HAL. Application can access device class specific features at the *Hardware Adaption Layer* or scarify performance and use platform independent functions from the *Hardware Interface Layer*. [6]

2.1.3 Hardware Abstraction on Embedded Systems

Hardware abstraction on embedded systems usually has to deal with limited resources in terms of computational power and memory. Thus concepts for hardware abstraction for the PC platform cannot be transferred one to one to embedded systems.

For wireless sensor nodes, one of the most resource restricted embedded systems, a flexible hardware abstraction with multiple levels of abstraction is proposed by Handziski et. al.. They use three layers for hardware abstraction, the two higher layers can be accessed by applications (see Figure 2.4). The bottom layer (*Hardware Presentation Layer*) which is closest to the HW is stateless and does little more than renaming HW resources. Register access and interrupts are transformed into more convenient function calls. The second layer (*Hardware Adaption Layer*) maintains state and hides the complexity of interacting with the HW. However specific features of the underlying platform (according to a certain device class) are exposed to applications. The top layer (*Hardware Interface Layer*) provides a platform independent interface. Capabilities of the underlying platform are upgraded (simulated in SW) or downgraded (hidden from applications). As the HW will evolve it will be necessary to introduce versioning for this layer. Application can access HW features at the second or third layer (counting started from HW nearest layer). Performance critical functions will sacrifice platform independence and use device class specific functions at the second layer while less critical function may use the third layer to become platform independent. [6]

2.1.4 Hardware Abstraction Revisited

Up to here we mainly used the same level of abstraction for all parts of our systems indifferent if they were hardware dependent or independent. An obvious approach to deal with hardware abstraction is to rise the general level of abstraction. If the application hasn't to be aware of the underlying HW platform it hasn't to deal with hardware abstraction.

Compiler for High Level Languages

One of the most prevalent methods for hardware abstraction, which many aren't even aware of, is the use of a High Level Language (HLL). For example, when an application is written in C++ and uses only built-in functions of the language or the standardized library, the goal of device independence is nearly perfect met. The application will be able to be deployed on every platform which is supported by the compiler.

During compilation for a specific platform, compilers usually first produce a platform agnostic intermediate code. This stage is not only used for optimization, it also allows the support multiple back ends. A back end is responsible for generating machine code and also contains platform specific functionality like platform specific optimizations. This approach is applicable to a lot of SW libraries which don't have to deal with HW issues. [7, Chapter 1]

Virtual Machines for Hardware Abstraction

The approach presented in the previous section has the big disadvantage that applications have to be compiled again before being deployed on a new platform. Another approach is to define a common virtual machine which applications can address. Then only the VM (or parts of it) have to be adopted to support a new platform.

A good example for this approach is the High-Level Language Virtual Machine (HLL-VM) (see Section 2.2.2). HLL-VMs overcome the problem of HW dependency by providing a virtual Instruction Set Architecture (ISA) which can be easily implemented on various HW platforms. Applications developed for a High-Level Language Virtual Machine (HLL-VM) usually only use resources provided by the VM framework (although in common implementations it's possible to access native functions too). Due to these properties the binary files can be moved to a different platform without changing them.

2.2 Virtual Machines

In this section we will take a closer look at VMs. After a short introduction to the general concepts for a VM we will focus on current trends in the development of VMs especially in the field of embedded systems.

2.2.1 Concepts

Definition: What is a Virtual Machine?

A definition of a VM is given by Popek and Goldberg [8]. They define a VM apart from pure SW emulators/simulators. A significant amount of the instruction of the virtual processor has to be executed directly on the real processor rather then in a software implementation. Formally they define this property as a mapping from the state of a real machine to the state of a virtual machine (and also an inverse must exist).

A more generally description is given by Smith and Nair, who describe a VM as a combination of hard- and software which runs a software in the same manner as on the platform it was developed for. The VM can offer different resources (in type and quantity) to the software running on it when compared to the hosting platform. [9, page 9]

Taxonomy

VM implementations can be separated in Process and System VMs. A Process VM is used to run user applications (thus process). Various implementations exists which can be divided in subcategories but all have in common to provide a virtual Application Binary Interface (ABI) to the application. A common type of process VMs are HLL-VMs like the Java VM or the .NET framework which are also called runtimes (see Section 2.2.2) . Also a classic OS can be seen as a process VM as it can run multiple application at the same time and virtualize HW resources for them. [9, page 13]

A System VM supports a whole OS which can on its side run multiple process. It can be placed directly on the HW or on top of an existing OS (hosted VM, e.g. VMWare). A System VM can also translate a virtual ISA to a different one which makes it possible to run a whole OS on a platform for which it wasn't developed for (emulator) or enable HW optimization without loosing compatibility (codesigned VM). [9, page 17]

Basic Techniques

A VM can be implemented in quite different ways, Figure 2.5 shows some basic types. The straight forward method is the **decode-and-dispatch interpretation**. In this case, a single dispatch loop performs a lookup for each instruction in the source code[1] to find the appropriate interpreter routine. This routine is invoked, executes the instruction and then returns to the main loop. This approach has nearly no overhead in terms of memory and startup latency but is very slow because all instructions are implemented in SW and each instructions needs a table lookup and procedure invocation. A small improvement is **threaded interpretations**. In this case no dispatch loop exists, instead the code for finding the next instruction is placed directly at the end of the interpreter routines. This removes the table lookup which saves execution time by using moderate more memory for the additional code.

More sophisticated methods include **precoding** and **binary translation**. Precoding transform the initial source code an intermediate form which is better suited for emulation but interpretation routines are still needed. In the case of binary translation, each instruction in the source code is translated into an instruction that can be directly executed on the target platform, no interpretation routine is needed. While this is obviously the fastest way to execute applications on a VM, it has considerable drawbacks in terms of startup latency, memory usage and also portability. [9, Chapter 2]

[1]In compliance with the referenced booked, source code is used here to describe any input code to the VM. This can also be a binary executable for an arbitrary ISA.

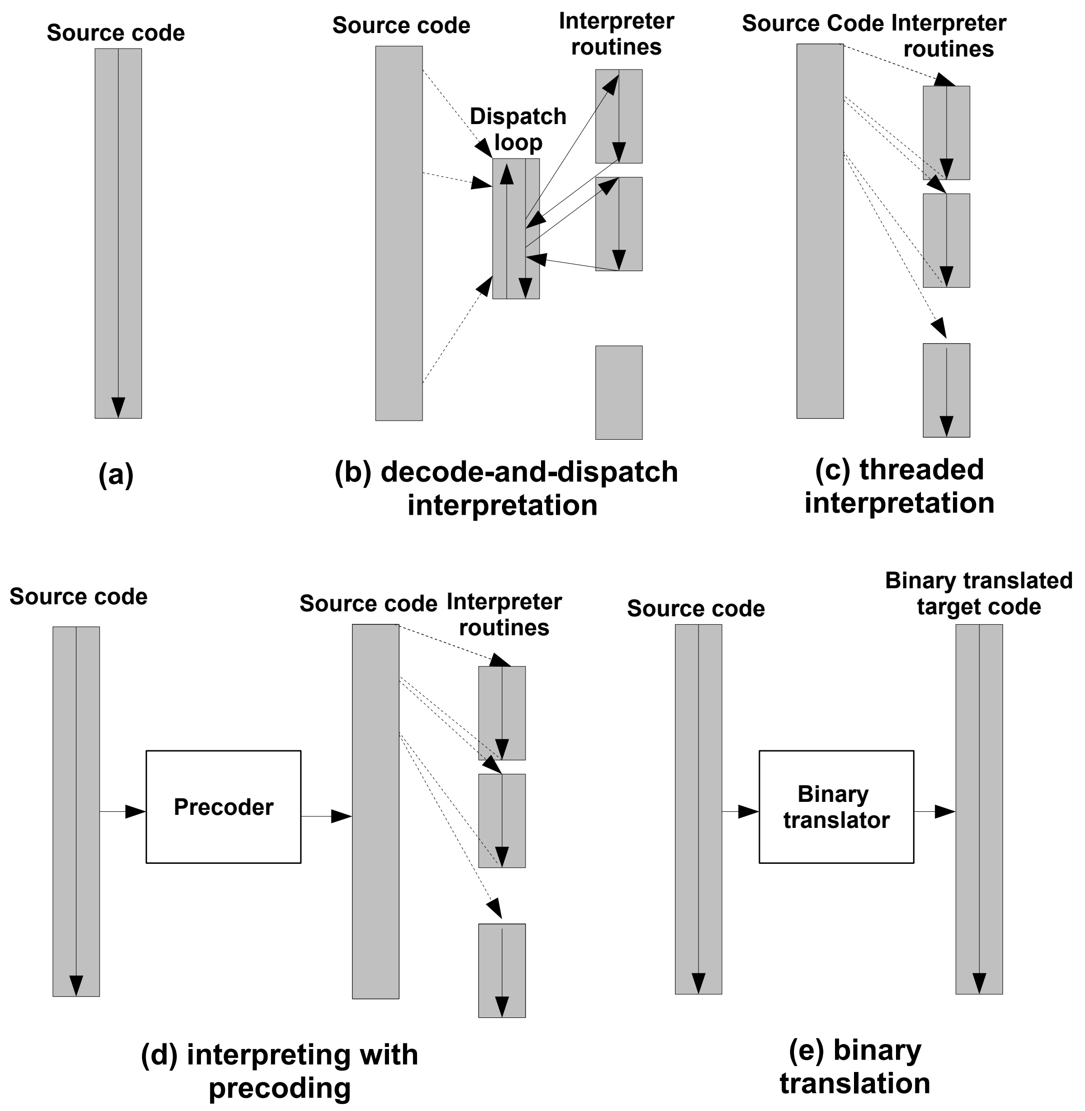


Figure 2.5: This figure shows different possible emulation methods for VMs. They range from pure interpretations (b and c) to partial (d) or complete (e) translation of the input code. All approaches have different tradeoffs regarding startup delay, runtime delay, additional memory usage or portability. [9, page 80]

As seen on the above described methods, where is always a tradeoff between memory usage, startup latency, execution time and portability. Not all VM place the same emphasis on all attributes. For example, an embedded VM will be more critical in terms of memory usage. On the other side a server VM will focus more on performance and for a code signed VM portability will may be even irrelevant. Often VMs have to deal with all four attributes. In this case, a combination of the described emulation methods is used (called *staged emulation*). Usually, at the beginning all code is interpreted. After a while, when some profiling data (eg. how often a block is executed) is collected the VM framework choses some code blocks to be optimized. In general four stages of can be distinguished: interpretation, basic translation, optimized block, highly optimized block. Every stage includes more sophisticated optimization methods, needs more profiling data and also introduces more overhead. Highly optimized blocks for example are usually only used in very long running applications. Some VM implementations may also skip a complete stage because it brings no benefit or can't be implemented due HW resource restrictions. [9, Chapter 3]

Optimization methods include reordering code blocks to support efficient caching. Another method is to inline procedures to avoid the overhead of a function call (although this means additional memory is used, it's reasonable for heavy used functions). Further optimization can be done by constant propagation (replace variables with constants if possible) and dead code elimination, although this is usually already done at compile time. Generally the optimization framework has to rely on profiling information gained from compiled code and doesn't has access to HLL information which is disadvantage compared to HLL compilers. HLL-VM overcome this problem with extensive use of meta data in the byte code to provide more information for optimizing. [9, Chapter 4]

2.2.2 High Level Language Virtual Machines

A big part of the success of VMs nowadays can be attributed to so called High Level Language Virtual Machines. As mentioned earlier the two main exponents this approach are the Oracle Java VM (original developed by Sun) and the Microsoft .Net framework. Both are popular SW developing tools and even expand to new areas (see Java Card, Section 2.3) which were before dominated by native programming languages (mostly C).

HLL-VM have in common that their ISA (and subsequently the byte code) is designed explicitly to be executed on a VM, therefore also called Virtual-ISA. This easies the development of the VM because it doesn't have to simulate the behavior of a silicon chip and the abstraction can be raised to a higher level. Because a HLL-VM is a special form of a process VM, it doesn't has to support running a whole OS which also reduces the functionality needed in the ISA. Functions like I/O and access to HW resources are instead moved to libraries. As the name implies the byte code is generated by compiling one (Java VM[2]) or more (.Net) mostly object oriented HLL. [9, Capter 5 and 6]

Java

The Java VM provides the target platform for the Java language. Java is a general purpose, concurrent and object oriented language which borrows heavily from C/C++. To reduce error-proneness some C++ features were removed, like pointer arithmetics. It also introduces new features like garbage collection for unused objects. Java was initially developed for customer electronic devices, therefore it had to be able to run on a lot of different HW platforms. The Java VM doesn't need to know anything about the Java HLL, instead it operates on an intermediate form referred as byte code. The byte code can be executed using various methods like interpretation or binary translation. [10, Chapter 1]

All information the VM needs are encodes in so called class files (are not necessary files, also this is the usual the case). Each class file represents one class or interface. Type checking is usually done before runtime and the distinction is made through type specific instruction (for example there are add instructions for each appropriate data type like integer, short or floating point types). The Virtual-ISA for the Java byte code represents a stack machine, which can be easily emulated on an arbitrary existing ISA. The published specification for the Java VM leaves a lot of design decision to the implementer. For example it is not specified which garbage collection algorithm should be used nor any method for optimization to improve speed or memory usage. Rather then that, an abstract memory layout is defined which consists of:

- **Programm counter** (per thread)
  Points to currently executed instruction.
- **Java Virtual Machine Stacks** (per thread)
  Stores Java VM frames. Frames are pushed/popped when a method is invoked/returned. Each frame holds an operand stack, a field for local variables and a reference to run-time constant pool.
- **Heap** (shared)
  It holds all dynamically allocated object (class instances or arrays). It's usually garbage collected, although implementations can decide not todo so.
- **Method Area** (shared)
  Comparable to the text segment in native executables. Contains the bytecode.

[2]Although multiple HLL can be used, Java is clearly the programming language of choice.

- **Runtime Constant Pool** (shared)
  Can be compared to symbol tables in other programming languages. Holds constants an reference to various fields/methods.
- **Native Method Stack** (shared)
  Used to support native methods which are implemented in a different language than Java (usually C/C++).

Java also support exceptions on byte code level. They can be generated by an explicit command (*athrow*), caused by an abnormal execution condition or by an error in the VM (eg. running out of memory). [10, Chapter 2]

.Net

The .Net platform is a continuation of the COM (Component Object Model) platform. COM focuses on contracts between individual programs which are expressed as type definitions. Microsoft describes COM as a programming model and a supporting platform. However COM has some serious drawbacks. There is no standardized way to describe contract definitions, two, only partial compatible, formats exists: Interface Definition Language (IDL) and Type Library (TLB). There is no way to express dependencies of components and the type definitions are platform dependent. The interchange between components depends on exact knowledge of vtables and stack layout at compile time. This make deployment on different platforms difficult.

To overcome this problems Microsoft introduced the Common Language Runtime (CLR). CLR is an implementation of Common Language Infrastructure (CLI) specification which Microsoft submitted to the ECMA for standardization. CLI consists of the Common Type System (CTS), Common Intermediate Language (CIL), file and metadata formats. Contracts for components of the CLR are described by using metadata, which removes the need to take care of platform peculiarities. Not until a CLR component is deployed or loaded, native binary code is produced. This includes translation of CIL to native program code. Microsoft claims reach the performance of native developed SW.

Applications for .Net can are usually developed in C#, a language developed by Microsoft which borrows heavily from C/C++ and Java. Theoretically any programming language is supported for which a compiler exists that can produce valid CIL code plus metadata. Practically a small number of other languages is supported, like C++ and Visual Basic. The normal execution mode for .Net applications is called managed execution which means that applications can be verified during runtime by the CLR. Verification include monitoring of variables and memory. Developers are encouraged not to manage memory or threads by themselves and should instead rely on built in functionality. [11, Chapter 1]

2.2.3 Low Level Virtual Machine

Another emerging field of applications for VM technologies is the Low Level Virtual Machine (LLVM). LLVM is a compiler frame work for code optimizations at all stages (compile time, link time, runtime etc.). Most of the optimization techniques use an intermediate representation which reassembles a RISC like ISA. In the following we will look at this code representation and show how it can be used to increase portability.

LLVM defines an instruction set consisting only of 31 opcodes and features a pure load and store architecture. All logical and arithmetic opcodes use the three-address form (take one ore two operands and store result at given position) and can be overloaded (can be called with different data types). LLVM is type safe and features four primitive and four derived data types. The primitive types are *bool*, *integer*, *floating point* and *void*. The numerical types also support different sizes (8-64 bit, single/double precision). Derived types are *pointers*, *arrays*, *structures* and *functions*. The authors claim to be able to reproduce HLL concepts like object orientation with this types. As LLVM is a type safe language type conversion is only possible via a *cast* instruction. All memory allocation is explicit through opcodes (even for stack variables). Function calls are implemented by a typed function pointer. LLVM also supports exception handling of HLL via the two opcodes *invoke* and *unwind*. The first defines a code block to unwind the stack and the latter initiates this process.

According to the authors, the key point of the LLVM intermediate language is on the one side to preserve high level information for optimization purpose and on the other side to stay low level enough to support arbitrary programs. [12]

emscripten

emscripten is a LLVM to Javascript compiler/translator proposed by Mozilla. The key goal is to allow arbitrary code which can be compiled to LLVM to be run in a Javascript VM which is present in nearly every modern web browser. The challenge is to transform a low-level language (LLVM) to high-level language (JavaScript) without forfeiting to much performance. As a solution, several method are proposed to recover high-level constructs from the low-level input. As an example the *relooper* is described, which constructs loops in JavaScript out of the branch operations of LLVM. [13]

PNaCl

Another approach for portable executables is the *Portable Native client* proposed by *Google*. The executables are distributed in the LLVM intermediate code and are translated to the target ISA on the client. As this is also a web centric approach security is a vital aspect. Google [14] has previously already proposed techniques to sandbox native code which are now also used. The final goal is to provide the performance of native code together with the portability and security of interpreted code. [15]

2.2.4 VMs on Embedded Systems

Embedded Systems pose a special challenge for VM developers. They are designed to fulfill a certain task and often have limited hardware resources. Several attempts were made to bring VMs to such platforms.

Most of them have in common that they use a custom byte-code. Some of them are derived from existing general purpose VM byte codes. An example for this approach is *Mote Runner*, which is designed to support multiple HLL and currently supports Java and .Net. It is a stack based VM targeted at Wireless Sensor Nodes. Due to the domain specific needs, it's centered around a reactive programming model, which means that if an event occurs a appropriate function is called. To save resources a concept similar to function pointer is used (called *Delegate* by the authors) instead of interfaces or abstract classes. The use of some language features is restricted, non existing but embedded system specific features can be added to the HLL via annotations. *Mote Runner* also permits the use of the language specify standard compiler which makes integration in existing development environments easy. The generated byte code is then translated into the custom byte code. [16]

A similar approach is taken by Shaylor et. al. for their VM. They use Java byte code based on the CLDC standard (Java for mobile phones). Their main target is to minimize the size of the byte code so it can be used in devices with very little memory. To achieve this, they reduced the operand fields in the byte codes and variables are explicitly typed. This makes it possible to remove type specific load/store instructions. The regained opcodes are used for more load/store instructions with included index to save an additional opcode. Further improvements include resolving references at installation time to abandon the symbolic constant pool and optimization of object creation. Instead of creating an object and initialize it separately which leads to the need of handling uninitialized objects, objects are created when initialized. [17]

VMs can also be built on top of an embedded OS. Examples are the *Mate VM* and *TinyReef* which both use custom byte codes. The latter is a register based VM, which is claimed to be more efficient in terms of computational efficiency (earnings through faster execution are higher than losses through larger memory footprint). TinyReef uses a fixed size 32-bit instructions where the first eight bit are the opcode and the rest serves as operand. [18]

The *Mate VM* is also tailored for sensor networks and built on top of TinyOS. Its design goal is to support very compact but functionally rich applications. To achieve this, powerful byte code commands are supplied such as a single instruction to sent a message. The VM is not intended to run any sophisticated computation, if an application needs this, custom byte codes can be defined and the needed functionality can be implemented in native code. The small size of the application allows to install them through the wireless network. *Mate* also supports application isolation on platforms there features like virtual memory or memory protection are not supported by the HW. [19]

An older but still interesting approach is given by Stanley-Marbell and Iftode with their *Scylla VM.* It's not a pure interpreters as most of the embedded VMs are but translates the byte code instruction directly to native instructions (called *on-the-fly-compilation*). It is designed as a virtual register machine so most byte code instructions can mapped one to one to a native instruction. The in instruction have variable length to save space. Scylla can optionally be placed on top of an OS, nethertheless an application only sees the VM interface. The VM also introduce memory checks to the native code, either static at compile time or if this isn't possible through additional native code an runtime. [20]

2.3 The Java Card Platform

As this thesis deals with implementing a Java Card test platform this chapter will be dedicated mostly to the Java Card standard. At the beginning we will give a short introduction to smart cards in general. After this we will start exploring the Java Card specification beginning with the language subset used for Java Card applets. In the following we will look at the binary formats used for Java Card and will conclude with the runtime requirements.

2.3.1 Smart Card Foundations

Smart cards have evolved from plastic identification cards. This cards were first used in the 1950s for financial transactions (credit card) in the US. They had only very simple security features like a reference signature or secure printings. With greater and worldwide distribution of credit cards, fraud and counterfeit became a seriously problem.

To overcome the security problems, but also to improve efficiency, magnetic stripe cards were introduced. The machine-readable information on the card made it possible to perform additional user verification (eg. the well known combination of card and PIN) and also allows complete electronic transactions without paperwork.

With the major improvements in the semiconductor industry smart cards became feasible. The first smart cards were memory cards with simple logic circuits to prevent writing data to certain memory areas. They were used as telephone prepaid cards in the era of phone boxes and later as health insurance cards. The next step were microprocessor cards, which could support arbitrary applications only limited by memory and computational power. With the availability of cryptography on smart cards, they were also introduced for financial transactions, first as custom solutions later in a standardized form (EMV standard).

Due to further improvements in the field of semiconductors, energy consumption of the chips was reduced to a level, contact less cards became possible. They drew power and communicate through an electric field emitted by the reader. [21, Chapter 1+2]

Hardware

Smart cards are usually using a custom System on Chip (SoC) consisting of an established micro controller Central Processing Unit (CPU), various memory types and additional HW components.

The CPUs used for smart cards usually have 8-bit word size but also 16-bit systems exist and even 32-bit CPUs are emerging. Most of the 8-bit CPU are based on the Intel 8051. Variants of this CPU are now available from various manufacturer, often with specific enhancements. The 16-bit CPUs are often advancements of the 8051. The 32-bit CPUs are usually based on an ARM core (ARM 7 or ARM Cortex).

Beside the familiar volatile RAM smart cards can feature multiple different memory types. Classic memory types to store information when the card is off power are ROM (read only memory) and EEPROM (electrical erasable read only memory). ROM is usually used for basic software routines which will never change in the lifetime of a smart card. Application data, but also application code, which has to be modified in the field, is placed in the EEPROM. A later introduced memory type is flash, which has many similarities with EEPROM and will may replace other nonvolatile memory types in future. Chip designer always have to make tradeoffs when selecting memory types for a smart card as they differ in attributes like access time and occupied chip area. RAM is always the fastest accessible memory but needs the most chip area per storage unit. ROM needs the least chip area per storage unit but has to be recorded during chip fabrication. EEPROM and flash are in between regarding the chip area per storage unit and have longer access delays.

Additional HW packed into the SoC can vary a lot. Common extensions are memory management units which allow separating memory areas to isolate applications, I/O components or cryptographic extensions. Modern smart cards support various symmetric key and public key algorithm. [21, Chapter 5]

Software

Software development for smart cards is done using assembler languages and C, with shifting more and more parts to C. Early smart cards contained only little customized code but soon some frequently used functions were outsourced into libraries. The libraries could be placed in ROM because they hadn't to be changed very often and could be reused for different use cases. This were the predecessors of the current available Card Operating System (COS). The available COS cover a broad range, reaching from very tiny systems, not much more than the initially described library collection, on 8-bit CPUs to mighty multi tasking and network capable systems on 32-bit CPUs.

Originally it wasn't planned to load additional program code onto smart cards after installing the basic systems. In recent years, a change of thinking occurred. The motivation behind isn't completely clear, but a strong argument is to fix bugs after release. The easiest way would be to load native code in the EEPROM or a similar nonvolatile writeable memory. However, in the world of smart cards that approach has several obstacles. First , as mentioned in the previous section, their are a lot of different HW platforms which prevents writing and distributing portable native code. Second and maybe even more severe, native code poses a security thread without further adaptions. This adaptions would have to be a sort of process isolation including a different execution mode for the CPU and memory protection. All things that are available on other platforms, like PCs, but usually not on smart cards. Because of these obstacles, interpreted code becomes a viable alternative. Despite the drawbacks of interpretation (slower execution and higher memory demand), it's now used in two common open[3] platforms: Multos and Java Card. Interpretation overcomes the two stated problems because the foreign byte code doesn't have direct access to the HW, instead it's interpreted and checked by an on card runtime environment. [21, Chapter 13]

Communication

Communication between Card Acceptance Device (CAD) and smart card follows a master-slave principle, it's always initiated by the CAD. After the smart card is inserted into a CAD it's turned on (the contacts have to be enabled in the right order). When initialized the card sends an Answer-to-Reset (ATR) to the CAD, which informs the CAD about basic properties of the smart card (like supported transmission protocols). After receiving an ATR the CAD can send a Protocol-Parameter-Select to adjust several settings of the smart card or can start directly by sending commands to the smart.

Commands are encoded as Application Protocol Data Unit (APDU)s. Two different forms exists, a command APDU is sent by the CAD to the smart card and a response APDU is returned when the card has finished its work (see Figure 2.6).

The command APDU consists of a header and a body. The class byte (CLA) selects a command set (e.g. '8X' for custom non standardized command set). The instruction byte

[3]According to Rankl and Effing, open is used here to describe smart cards which allow the execution of foreign code. It's not used in term of open source SW like Linux.

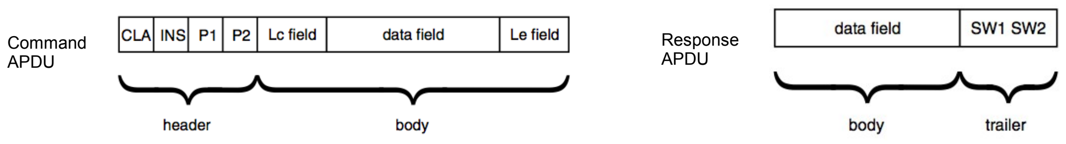


Figure 2.6: This figure shows structure of an APDU. Parts of the figure are carried over from Rankl and Effing. [21, Chapter 8].

(*INS*) is used to select a specific command which can be augmented with two parameter bytes (*P1* and *P2*). Following the header comes an optional body. It consists of an length field (*Lc*) for the following variable length data section (*data*) and a field with specifies the expected length of the response (*Le*).

The response APDU consists of a data field (the length was specified in the command APDU) and a status word consisting of two bytes (*SW1* and *SW2*). The status word indicates if an error occurred during processing (otherwise '9000' is returned which stands for 'normal processing'). [21, Chapter 8]

2.3.2 Java Card Language Subset

Java Card is based on the Java SE and shares many properties. Despite, full support of all features isn't possible on resource constraint devices found in smart cards. Therefore a subset was defined which ensures compatibility with the Java SE and doesn't overwork existing resources. Compatibility of course works only in one direction, Java Card applets can be compiled and executed using Java SE, not otherwise. The subset was carefully chosen to preserve the main attribute of the Java language, like objects and inheritance. For supported, optionally supported and not supported features see Table 2.1.

As security is one of the main aspects of Java its also an important point for Java Card. To save resources on the card device, the byte code verification can be done off device, the JCRE just have to load the CAP file and execute. On device, byte code verification, either during loading or at runtime is possible but not mandatory. Security on applet level is provided by a context assigned to each object. Every applet and also the JCRE possess its own context. When an applet is installed and creates an object, the object gets assigned the context of that applet.

Java Card supports exception handling much the same as Java SE. A major difference shows in how to handle uncaught exceptions. The Java SE VM will just hold the concerning thread. As the Java Card VM is single threaded the whole VM will halt. What happens next isn't specified in standard and differs across implementations.

Supported	**Optional Supported**	**Not Supported**
Packages	Integer Data Type	Dynamic Class Loading
Dynamic Object Creation	Object Deletion Mechanism	Security Manager
Virtual Methods		Finalization
Exceptions		Threads
Interfaces		Cloning
Generics		Access Control in Java Packages
Static Import		Typesafe Enums
Runtime Invisible Metadata		Enhanced for Loop
		Varargs
		Runtime Visible Metadata
		Assertions

Table 2.1: Overview of Java language features supported, optionally supported or not supported by the Java Card Platform. Comprehensive information about the features can be found in [22, JCVM Spec Classic Chapter 2.2].

Java Card supports a slight different set of data types than Java SE. Byte and short are fully supported, support for integer is optional and floating point data types are not supported. Java Card supports the reference type identical to Java SE. Arrays can only be one-dimensional. Data types in Java Card use an abstract storage unit to indicate the size of a data type, called word. A word must be capable of holding the value of type byte, short and reference. Two words must be able to store an integer. [22, Chapter 1 to 3]

2.3.3 Java Card Binary Format

Application developed in Java are usually (e.g. for Java SE) stored in binary files, called *class* files. They consist not only of the byte code but also of interface information (types and names) of the containing classes.

Java Card applications are first also compiled to class files, which allows to use the standard Java compiler and all of its advantages like code checking and optimization. In the next step the class files of a whole Java package are converted into a Java Card package. For a successful conversion, only the language subset related to Java Card is allowed as prior input for the compiler. The obtained file are the Java Card converted applet (CAP) file and export file. We will explain them in the following together with the so called application identifier (AID), which is used to identify applets.

The CAP File

Java Card packages are delivered in the so called CAP format. A CAP consists of a set of components stored in a JAR file. Each CAP component is a stream of bytes stored in big endian order. The first byte is a token which identifies the component type. Table 2.2

shows all standard CAP files. A developer can define custom components which must be tagged ranging from 128 to 255.

Java Card separates two types of packages: applet and library. Applet packages contain at least one class which inherits from *Applet*. Thus it also has an Applet component (see listing below) which is used to install it on a device (see Section 2.3.4 for details).

Component Type	**Tag Value**	**File Name**	**Remark**
COMPONENT_Header	1	Header.cap	
COMPONENT_Directory	2	Directory.cap	
COMPONENT_Applet	3	Applet.cap	optional
COMPONENT_Import	4	Import.cap	
COMPONENT_ConstantPool	5	ConstantPool.cap	
COMPONENT_Class	6	Class.cap	
COMPONENT_Method	7	Method.cap	
COMPONENT_StaticField	8	StaticField.cap	
COMPONENT_ReferenceLocation	9	RefLocation.cap	
COMPONENT_Export	10	Export.cap	optional
COMPONENT_Descriptor	11	Descriptor.cap	
COMPONENT_Debug	12	Debug.cap	optional

Table 2.2: This table shows the standard components of a CAP file. They are store in a JAR file. The first byte of a component is the tag which identifies it.

The most important components when developing a Java Card VM are described in the following:

- **Constant Pool Component**
  The Constant Pool component contains reference to all methods, fields and classes used in this package. When a byte code instruction accesses any of these types it starts with an index into the Constant Pool. The index points to a info field, which is always four byte long. The fixed size allows direct access by calculating an offset. If the class, field or method requested is defined inside the current package, the Constant Pool field contains an offset into the Class (for classes, virtual methods and instance fields), static field image (for static fields) or Method component (for static or private methods). If the requested type is defined outside the current package the Constant Pool entry references the Import component by tokens.

- **Class Component**
  The Class component describes all classes defined in this package. It holds sufficient information to create instances, to perform method or field access (method tables) and to check cast operations (class hierarchy). For method invocation it references

to Method component. External superclasses are referenced through the Import component.

- **Method Component**
  The Method component contains the byte code for all methods defined in this package. It also includes information necessary to invoke a method, like number of parameters, number of local variables and the needed stack size.

  The byte code featured in this component is also a subset off the original Java byte code. For example it lacks functions for floating point arithmetics and concurrent execution management.

- **Staticfield Component**
  The Staticfield component is used to initialize all static fields defined in this package. In the Java SE this work is done by the `clinit` method. Static fields are store in a *static field image*, also offsets in the Constant Pool component reference there.

- **Export and Import Components**
  These components are used to share resources across package (CAP file) borders. The import components defines packages, which are used in the current package. The external packages are identified by their AID. When the package managers finds the corresponding CAP file it can use the Export Component to find the wanted class or method.

  The Export component describes all classes, static functions or fields which are offered to other packages (this means they have to be public). Static fields and methods can be accessed directly by using an offset into the static field image respectively the *Method* component. Virtual methods have to be searched in the method table of the corresponding class.

- **Header and Applet Component**
  These components are used to identify the package (*Header* component) or applets provided by this package (*Applet* component) by their AIDs. The Header component additional contains a major and a minor version number and various flags. An applet additionally has a reference to the *Method* component to invoke the `install` method.

The references between components of one package and entry/exit points for references between packages are shown in Figure 2.7. During execution, all classes, fields or methods are addressed through the ConstantPool. If the requested entity is in the same package the ConstantPool supplies an offset to access it directly, otherwise if it's in another package, the Import Component supplies the AID of the corresponding package. Other packages can refer to entities of this package through the Export Component. [22, Chapter 6]

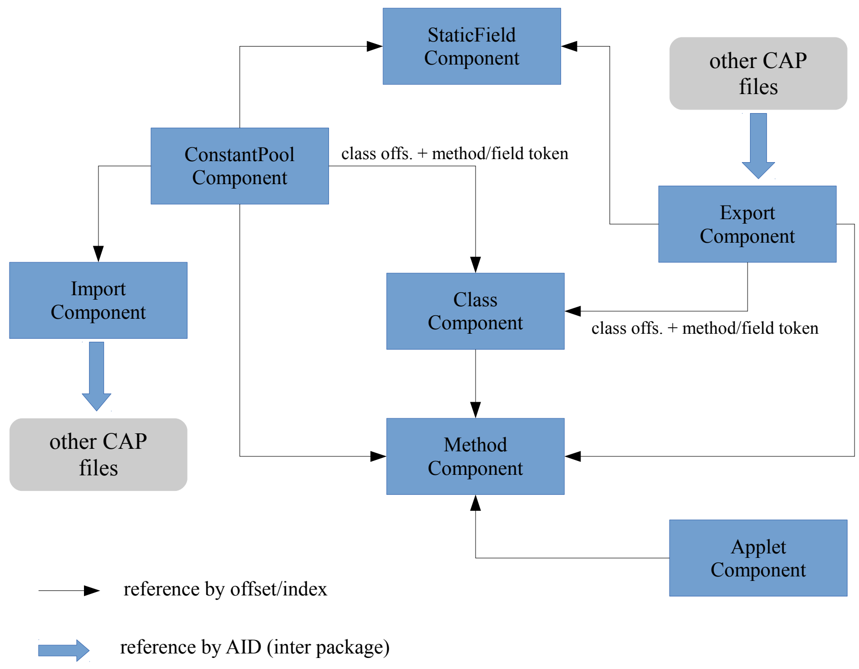


Figure 2.7: This figure shows the references between CAP components. Starting point is either the Constant Pool component if this is the current package or the Export component if this package is accessed from outside (another package). When this package is deployed on a device the Applet component directs to the install method for each application.

The Export File

The export file contains information about the public accessible API of a whole package. It can be used to convert a package into a CAP file that is binary compatible with existing applications which are linked to the existing export file. [22, Chapter 5]

AID - Application Identifier

AID is the mechanism for naming applets and packages in Java Card. It is defined in ISO 7816-5. An AID consists of a 5-bit RID (resource identifier), which is assigned to organizations by the International Organization for Standardization (ISO) and a 0- to 11-bit PIX (proprietary identifier extension) managed by the related organization. [22, Chapter 4.2]

2.3.4 Java Card Runtime Enviroment

The Java Card Runtime Enviroment (JCRE) consists of the Java Card Virtual Machine (JCVM) and the Java Card API classes. We will take only a short look on JCVM in this chapter and will focus on the functionality provided by the API.

The demands for the VM can be deduced from the binary representation of an applet described in the last section. In Section 3.2 we will discuss a system architecture that fits the demands of the Java Card specification and also fulfills our design goals.

In the following we will examine the major runtime functionalities, from installing applets over to how to select and invoke an applet, to how the JCRE supports the applets doing their tasks. At the end we will look at security an error recovering mechanism.

Installing Applets

Applets are installed on a Java Card capable smart card by calling the static method `Applet.install`. The function is implemented in the JCRE class but throws an exception and has to be overwritten by an applet. When called, the method has to create an instance of its associated class and register it in in the JCRE via calling `Applet.register`. The applet deems to be installed correctly when `install` calls `register` and no exception occurred. `register` is a method supplied by the JCRE which assigns the committed AID to the now installed applet. [23, Chapter 3]

Applets and libraries and can be put on the smart card during production process (then it's also possible to place them completely in the ROM) or loaded afterwards. In the latter case an installer is needed on the card device. The Java Card specification doesn't requests an installer but implementers are free to provide one or multiple installer. Towards the CAD the installer behaves like a normal applet and communicates via APDUs. It's up to the implementer to decide if he want's to implement an installer in Java (and thereby as an applet with special privileges) or in native code (and thereby part of the JCVM). [23, Chapter 11]

Selecting and Invoking Applets

The CAD can select an applet by using special command APDUs (*SELECT FILE* or *MANAGE CHANNEL*) and supply a valid AID. To select an applet the JCRE calls `Applet.select`, a method that should be overwritten by the implementing class. This method can do some initialization work and has to return a boolean value, true if selection is successful or false if the applet denied being selected. When an applet is successfully selected the current context is set to the context which belongs to the applet.

After finishing `Applet.select`, the JCRE calls `Applet.process` (also a method that should be overwritten by the implementing class) and forwards the received APDU. The

APDU used to select an applet is also forwarded to this function. APDUs are encapsulated in a Java class (`APDU`) which allows to access fields with convenient getter and setter methods. The current applet is deselected when a new applet is selected. `Applet.deselect` is called which allows the applet to do some cleanup. [23, Chapter 4]

API Support

The JCRE supports the applet with a lot of built in methods. They are grouped in several packages. A subset of the JDK classes can be found in the packages `java.io`, `java.lang` and `java.rmi`. They define classes which are well known from other Java editions like the ubiquitous *Object* or a number of possible exceptions. Java card specific methods can be found in `javacard.framework` and `javacard.security`. The first package defines classes for basic functionality, like the abstract class `Applet` which every Java Card applet inherits from, classes to deal with APDUs and AIDs and several utility methods (class `Util`). Extensions can be found in the package path `javacardx.*`. They range from extended length APDU over classes to access additional memory to extensions of the utility methods provided in `javacard.framework`.

The JCRE provides applets a transaction mechanism. An applet can start a transaction by calling `JCSystem.beginTransaction` and finish it by calling either `JCSystem.commitTransaction` (modifications are made persistent) or `JCSystem.abortTransaction` (modifications are discarded). A transaction always ends if the JCRE gains back control (by exiting one of the methods defined in *Applet*) .The JCRE doesn't support nested transaction and has, due to the limited resources of a smart card, also a limited capacity of operations a transaction can contain. [23, Chapter 7]

Security for Applets

Additional to the Java language security features (e.g. type checking or protection attributes) the JCRE has a firewall which prevents applets from interfering each other. The specification requires a certain minimum of runtime checks but implementors are free to include more checks as long they are transparent to applets.

As mentioned earlier, the firewall separates applets using a different context. Every package, and the JCRE itself, has its own context. Applets from the same package also have the same context. Every object gets the context assigned, which is active when the object is created. Generally, access is only allowed within the same context. Context violation is checked every time a sensitive byte code (all byte codes that access fields or arrays or which invokes instance methods) is used.

To allow information interchange between applets where are several exceptions to this general rule. First the JCRE works as a sort of system applet which has access to all existing context. This is also necessary to invoke functions like `Applet.process`. The over way around are the *Java Card RE Entry Point Objects* which can be used to invoke methods of the JCRE from any context. Through this, applets can use the functionality provided by the JCRE. Exceptions are also made for global arrays and shareable interfaces. An interface is made shareable through implementing *javacard.framework.Shareable*. [23, Chapter 6]

2.4 Summary

In this chapter we gave an introduction to hardware abstraction, virtual machine design and the Java Card platform.

We saw that hardware abstraction can be basically done straight forward and is heavily used on the PC platform. Through abstraction usually adds some overhead, problems occur when hardware abstraction should be done on resource constrained platforms. We also saw that compiler and virtual machines provide a good hardware abstraction for functions which don't rely on special hardware features.

In the following we learned something about the basics of virtual machine, including some slightly different definitions and a taxonomy. We saw that designing a virtual machine is always a tradeoff between startup time delay, memory usage, execution delay and the degree of platform independence. We looked at some of the most prevalent VM platforms (Java and .Net) and also introduced the quiet new LLVM compiler framwork. We presented some approaches to bring VMs to very resource constrained devices, like Wireless Sensor Nodes, which needs some adaptions and tradeoffs.

Finally we gave an introduction to the Java Card platform, preceded with an overview of the hardware, software and communication protocols used for smart cards.

Chapter 3

Concept and Design

This chapter should give a better insight to the ideas behind the implementation. We will divide the chapter in three main parts. The first will deal with the overall design goals. This will lead us directly to the second part, where we will explain the main decision for the SW architecture. The last part will deal with the toolchain we select to achieve our design goals.

3.1 Design Goals

The overall goal of this thesis is to create the SW part of an test framework for new features for JavaCard based smart cards. It is NOT intended for use in a productive environment. For this case a lot of commercial (and thus usually closed source) implementations and the reference implementation exists. As we have different goals, our implementation won't compete with the existing ones in term of performance and will lack some optimization work usually done by them. Instead of optimizing our VM to a certain platform we have identified three different main goals: **Compliance**, **Portability** and **Maintainability**.

3.1.1 Compliance

To get meaningful test results the VM should be compliant to the Java Card standard[1] We don't want to mimic an existing implementation in all details (even not the reference implementation) as we omit several platform specific performance tweaks and don't implement runtime checks not demanded by the standard.

As the standard leaves a lot of design decisions to the implementor we strive towards a lean implementation of the Java Card standard. We will comply to the standard where

[1]The standard is defined in the *Virtual Machine Specification for the Java Card Platform*, the *Runtime Environment Specification for the Java Card Platform* and the *Application Programming Interface for the Java Card Platform*. We will use version 3.0.4 of the classic edition. All documents are available under `http://www.oracle.com/technetwork/java/javacard/specs-jsp-136430.html`.

it explicitly demands something and will rely on testing our implementation against the reference implementation for not explicitly stated cases.

3.1.2 Portability

An important design goal is to support multiple HW platforms. According to Brown [24] this attribute can be called *Portability.* The targeted platforms can range from a 64-bit x86 architecture running a POSIX compatible OS (as a platform for SW development) down to a 8051 compatible micro controller (for testing real world Java Card applications). Porting to a different platform should require as few changes as possible in the source code. Especially micro controller platforms are often only supported by few or even only one compiler so the source code should be compatible with all (or at least a large set) of currently available compilers.

To achieve this goal it will may be necessary to make tradeoffs in non-functional attributes. Platform specific optimization (e.g. for specific memory architectures) is only possible to a limited extend, as it would require to maintain additional code for each platform. In such cases we will aim on preserving portability at lowest possible cost (in terms of computational and memory efficiency).

3.1.3 Maintainability

The source code written during this thesis should be maintainable in multiple ways. Beside the classic task to correct programming errors it should also be easy to modify and/or to extend the source code. This target may comes in conflict with writing compact and efficient code. In such cases we will focus more on easy readable code, which is in our opinion the key factor for maintainability, and lesser on compact and efficient code. To compensate drawbacks from this strategy we rely on compiler intern optimization methods and will try encourage them in the source code (eg. inlining for small helper functions).

3.2 System Architecture

A quick outline of our system architecture can be seen in Figure 3.1. We decided to use a three layered architecture to encapsulate different kinds of functionality. Interaction between the layers is only done through well defined interfaces. Only the lowest layer is platform dependent, the other can be ported by simply compiling them for the targeted platform. Therefore multiple instances of the HAL exists to serve different platforms, as it can be seen in the figure.

A layer is again divided into modules which are responsible for different tasks and encapsulate functionality. This separation is done for better maintainability although the separation is not so strict than between different layers. There is no externally defined interface between the modules of a single layer. Instead we try to keep interaction as small as possible but make tradeoffs if costs (in term of code size and runtime behavior) for abstraction are to high.

3.2.1 Hardware Abstraction Layer

The HAL consists of two parts: the interface (`hal_interface.h`) and multiple implementations for different platforms. The code size of the implementations depends largely on the specific platform. For example when running on top of a desktop OS it will be very small because all needed functionality is provided by simple library calls. When running on a micro controller like a 8051 derivative without an underlying OS, it will may be more code necessary to supply needed functionality or to startup the device.

HAL Interface

The interface specifies functions that can be used by the OS Layer. They can be largely grouped in the following categories.

- **Communication**
  Functions of this category are used to send and receive bytes. The HAL doesn't have any insight to the content of these byte streams. As Java Card only reacts to requests, the receive function can be invoked in blocking mode to wait until new data is available.

- **Memory Management**
  Functions of this category give the OS Layer information needed to access the memory. This consists of start and end address of memory areas for transient and persistent memory.

- **Boot**
  Specifies a function (`start_os`) to start the OS Layer. After all initialization work is done by the HAL, it calls this function to handover control to the OS Layer.

3.2.2 OS Layer

The OS Layer provides rudimentary functions of a conventional OS. They can be accessed through an interface (defined in `os_interface.h`). The implementation is divided into four modules for core functionality, communication, memory management and (closely coupled with memory management) transaction management.

OS Interface

The interface specifies functions that be used by the VM (or by any other potential native application). They can be largely grouped in the following categories.

- **Communication**
  The OS Layer provides functions (`read_capdu` and `send_rapdu`) to send and receive APDU messages. The function can be called in blocking mode.
- **Memory Management**
  The OS Layer provides function quite similar to the memory management functions provided by the C standard library. The available functions can allocate and free memory areas of arbitrary size (as long as enough memory is available), either in persistent (`jcos_malloc_persistent` and `jcos_free_persistent`) or transient (`jcos_malloc` and `jcos_free`) memory area.
- **Transaction Management**
  The OS Layer provides a rollback buffer which can be filled by calling `record_transaction` and supplying a memory area. When a transaction is finished, calling `commit_transaction` will make the changes permanent. Calling `rollback_transaction` will restore the the values present at calling `record_transaction`.

Core OS Modul

This modul (`os_layer.c`) takes over control from the HAL. It calls the initialization functions of the other modules. When the whole OS Layer is ready it hands over control to a native application (usually the VM).

Communication

This modul (`comm.c`) converts the byte stream received from the HAL interface to APDU a message (byte array). It also takes a whole APDU and transfers it byte wise using the HAL interface.

Memory Manager

The *Memory Manager* (`memm.c`) maintains a table for the transient and the persistent memory area. As these areas differ only in start and end address, the same data structures can be used (see Table 3.1).

To reduce memory segmentation the memory manager tries to recycle freed memory segments. When a segment is freed it's marked as inactive. If the requested segment fits in an inactive segment, the segment will be reused. The surplus memory (if there is any)

`memm_entry_s`			
Var Name	**Type**	**Size (bytes)**	**Remark**
`state`	enum	platform specific	`EMPTY, ACTIVE, INACTIVE`
`length`	uint8	1	
`data`	byte array	length	

Table 3.1: Data struct for memory memory entry.

remains unused. An optional feature will be defragmentation which will be invoked when a requested segment exceeds the size of continuous memory available.

Transaction Manager

The transaction manager maintains a fixed size buffer (the size of this field also defines the commit capacity) to store the original memory state. When a transaction is committed the buffer is cleared and nothing else has to be done. When a transaction is rolled back the transaction manager goes through the field and writes the values stored in the buffer to the associated memory areas.

`transm_entry_s`		
Var Name	**Type**	**Size (bytes)**
`ptr`	platform independed pointer	platform specific
`length`	uint8	1
`data`	byte array	length

Table 3.2: Data struct for transaction memory entry.

The records are organized using C structs (see Table 3.2) and stored continuously in a byte array. When a new record is added the transaction manager traverse the buffer from the beginning and checks if the specified memory is already buffered (same position and size, overlapping areas are treated as new ones).

The byte array is operated as a stack and grows towards its start address. In case of a rollback, the first entry is processed at last. This ensures that memory is restored to its original state as older records overwrite newer ones.

3.2.3 VM Layer

This layer implements a Java Card VM. It uses, aside from programming language built in functions, only functionality provided by the OS Layer. It provides all interfaces required by the Java Card specification. The implementation is divided into modules which are loosely coupled. The public functions of a module are declared in the common header file (`jcvm.h`). This header file also defines data structs which are used all over the VM layer (e.g.. structs to maintain class instances).

Core VM

This module (source file `jcvm.c`) implements the main functionality of the Java Card VM. It contains the dispatch loop and holds all the VM's runtime data (see Section 2.2.2). Other modules are initialized and provided with helper functions by this module. The OS Layer starts the VM by calling a function (`start_VM`) of the core VM module.

Bytecode Interpreter

This module (source file `bcode.c`) implements the interpreter functions for the Java Card byte codes. It provides a static array of function pointers which can be used to call interpreter functions by their index. Interpreter functions always exit with a result code. This code indicates if the instruction was executed correctly, further treatment by the core VM module is needed (eg. in case of return statements) or if an error occurred (eg. unsupported instruction or internal error). The byte code interpreter module is stateless, it uses only local variable inside functions.

Object Manager

The Object Manager (`objm.c`) is responsible for dynamically created objects (class instances and arrays) and static fields. He creates, initialize and provides function to access these objects and fields. Internally the three different kinds of objects are administered by specific C structs (see Section 3.3 for arrays and class instances, static field image is included in the CAP file struct).

Arrays can be created (`new_array`) with a given size and type. Individual elements can accessed (get and set functions, `*_array_field`) using the index of the element. Class instances can be created (`new_class_instance`) and instance fields can be accessed (get and set functions, `*_instance_field`), both by using an index into the Constant Pool Component (pointing at a `CONSTANT_Classref_info` or a `CONSTANT_InstanceFieldref_info` element). Static class fields are initialized (`init_static_field`) using the Static Field Component of the respective package and accessed (get and set functions, `*_static_field`) using an index into the Constant Pool Component (pointing at a `CONSTANT_StaticFieldref_info` element).

Java Card Runtime Component

This module provides the functionality defined in the Java Card Runtime Enviroment (JCRE) specification [25]. It basically consists of two parts, a set of Java packages and a C source file (`jcre.c`). The Java packages serve as front-end, which can be invoked by an applet, and will be included as CAP files. This module is described in more detail in Section 3.4.1.

Package Loader

This module (source file `package_loader.c`) allows to access information stored in the CAP file in a convenient way. It resolves references from the `Constant Pool Component` of the CAP file into structs for methods, fields, etc.. Structs to access CAP information are defined in a header file (`cap_data_structs.h`) which is also part of this module.

3.2.4 JCEP Lib

The files falling under this term are not really a distinct module. They are more a sort of library and language extensions to aid the goal of platform independence. The header file `jeep_common.c` defines various data types, which behave similar across different platforms (regarding size and numerical representation). Only primitive datatypes which are used in everyday programming (e.g.. pointer or signed numerical) are defined, no special purpose structs. The file will be included in all source files of the project.

The source file `jcos_lib.c` defines a rudimentary standard library to be independent from libraries supplied by the development environment. These functions are declared in the header file described above.

3.3 Data structs

Beside the system architecture, data formats, which are used to store information during execution, are a major design decision. In the following we will describe the most prevalent of them in more detail. We will implement them using C structs or unions. Several of them are shared across multiple modules, mostly in form of pointers.

3.3.1 VM Value

VM variables of all types (byte, short, reference) are stored in an union (called `value_u`) consisting of the respective C data types (8- and 16-bit integer and pointer). The union ensure platform independence as the size of a pointer varies across different platforms.

3.3.2 VM Frame

The VM frame, which is used to store runtime information, is represented using a struct called `vm_frame_s`. We already introduced it for Java SE in Section 2.2.2. Java Card uses only a subset of the frame elements from Java SE. On the other side we will add some elements to the VM frame, which we think are are closely related to the currently executed method, like the current context.

Table 3.3 shows and explains the content of the VM frame used in our implementation. `local_vars` and `op_stack` reference to the same memory area. Local variables and the operand stack can be placed in the same memory area in interleaved form as the length of both is known at compile time. Before a method invocation, arguments are pushed onto the stack and can subsequently be used as local variables.

`vm_frame_s`		
Name	**Type**	**Description)**
`local_vars`	`value_p`	local variables used inside a method
`op_stack`	`value_p`	operand stack used inside a method
`op_stack_base`	`value_p`	base pointer for operand stack
`vm_pc`	`ptr`	byte code program counter
`current_cap`	`cap_file_p`	CAP file this method belongs to
`current_context`	`cap_file_p`	currently active context (specified via package)

Table 3.3: Data struct for a VM frame.

3.3.3 VM Method

This struct represents a method which can be executed by this VM (see Table 3.4). The fields are extracted from the *Method Component* of the CAP file. Arguments and the return values are communicated through the operand stack.

`method_s`		
Name	**Type**	**Description)**
`max_stack`	`value_p`	stack size, read from CAP file
`flags`	`value_p`	method flags, read from CAP file
`max_locals`	`cap_file_p`	number of local variables, read from CAP file
`nargs`	`cap_file_p`	number of arguments, read from CAP file
`bcode`	`ptr`	pointer to byte code

Table 3.4: Data struct for a method. It contains all information the VM needs to execute it.

3.3.4 VM Object

This struct stores field values and meta data of a class instance or arrays (see Table 3.5). Meta data include the type (class object or primitive/reference type array), the length, the owning context and a number of flags. Field values can be an array of a specific type or the fields of a class instances. In the latter case the field is of type `value_u`.

obj_s		
Name	**Type**	**Description)**
`type`	`uint8`	object type according to VM specs
`field`	`ptr`	pointer to class fields or array
`length`	`uint16`	number of fields/array elements
`cap`	`cap_file_p`	type info (not used for primitive arrays)
`cinf`	`class_info_p`	type info (not used for primitive arrays)
`context`	`cap_file_p`	owning context
`super`	`obj_p`	pointer to super class (only for class instances)
`flags`	`obj_flags`	access flags

Table 3.5: Data struct for dynamically created VM objects.

3.4 Runtime Features

This section will describe some concepts for requested and more sophisticated features in more detail. It should be seen as an extension of the previous section. As the following features need support from multiple components they don't fit in the structure we used to explain the system architecture. Instead we will portion them in appropriate groups, state the requirements and show there and how they will be implemented.

3.4.1 Native Extensions

Requirements

Java Card provides rich functionalities for applets running on a smart card. The JCRE specification requests a set of mandatory features, which we already introduced in Section 2.3.4. A Java Card implementation must support them to comply with the Java Card specification. Some of this features are transparent to applets, but most of them have to be accessed through a set of Java classes.

Some methods in this libraries need access to VM resources which aren't exposed to usual applets, like access to I/O ports (included in `javacard.framework.APDU`) or to control transactions (included in `javacard.framework.JCSystem`).

Concept

To fulfill these requirements we will split this functionality into two parts. One part will be implemented in Java (in multiple packages aka CAP files) and the other part will be implemented in C (source file `jcre.c`). The structure and the way how method/function calls are forwarded can be seen in Figure 3.2.

Classes and methods which don't need access to any special resource will be implemented straight forward in Java. This includes interfaces, abstract classes and methods that do only computation on VM data types. They are converted using the export files delivered with the Java Card Development Kit (JCDK) to be binary compatible with existing applets.

Methods which need access to VM intrinsics have at least a Java proxy method. To switch to the native context, the JCRE method calls a corresponding static method in class `jcep.Native`. This will trigger the VM to execute the `invokestatic` byte code which can be intercepted. The VM will recognize an invoke on the class`jcep.Native`. This class doesn't contain any code, all methods are only stubs. We decided to introduce a second layer of static methods (the JCRE classes already consists mostly of static methods) where we can intercept method calls, because it makes the interception of method invocation much easier. Due to this we don't need to mark individual methods as native, instead we only have to watch for attempts to access a certain package.

The VM launches the byte code execution through a special static method (`CardMgr .main`). Before invoking this method, the initial VM frame is created. It holds the base pointer to the mixed operand/variable-stack. When creating a new frame the stack pointer will only be raised to reserve memory for local variables and the operand stack of the invoked method.

3.4.2 Card Manager

Requirements

When (re)starting a Java Card device, a default applet is required to be active. It should be capable of accepting the APDUs intended to select an installed applet. In the following it should select the requested applet by invoking `Applet.select` and forward incoming APDUs through calling `Applet.process`. In the case of selecting another applet during a single session it should also call `Applet.deselect`.

Additionally this applet should serve as an entry point for byte code execution. Together with the functions described in Section 3.4.1, it should serve as a connector between the Java and the native environment.

Concept

In our implementation, the *Card Managers* (class path `jcos.cardmanager.CardMgr`), together with some native helper functions (source file `jcre.c`), will do the above described job.

As a first step the OS Layer starts the VM by calling the C function `start_vm`. This function prepares the stack and hands it over to the initial method of the *Card Managers*. In the following the *Card Managers* waits for an incoming APDU by calling `Native.readAPDU`. This method triggers a call of the native function `read_apdu_native` which blocks execution of the VM until a full APDU is received through the *OS Interface*. The native function is supplied with a byte array, which is handed down to the OS Layer to serve as a buffer for the incoming APDU. Through this arrangement, no further memory or copying is needed.

The *Card Managers* maintains a list of all installed applets. When receiving an APDU, intended to select an applet, the *Card Managers* looks for an applet with the requested AID. If a suitable applet is found, the applet is selected using the methods defined in the applet super class. Figure 3.3 shows the relations between applets, the *Card Managers* and the other parts of the JCRE in a class diagram.

3.4.3 Transaction Management

Requirements

The Java Card platform supplies applets with a transaction mechanism. Applets can modify multiple fields during a transaction and rely on the fact that either all fields are modified correctly or all remain in their previous state. The amount of fields that can be modified during a transaction depends on the buffer capacity of the transaction mechanism. If a field is modified and the buffer is exceeded and the transaction is aborted. Section 2.3.4 describes transaction together with other JCRE capabilities.

All features stated above only refer to persistent variables which include arrays, instance fields and static fields. Local variables or the stack content are only temporary variables which are always deleted after each session and therefor are not included in the transaction mechanism.

Concept

Hereunder only a few byte codes (listed in Table 3.6) are critical for the transaction mechanism. Every time one of these instructions is executed the VM checks if a transaction is in progress (indicated through a global variable . In that case the transaction record function of the OS Layer is called and supplied with the appropriate memory location (see Section 3.2.2 for interface definitions and implementation).

Commits and rollbacks can be initiated through the class API. Automatic rollbacks will be initiated on deselecting an applet, when an uncaught exception occurs or on power-on-reset.

Byte Code	**Description**
`<t>astore`	store a single element in an array
`putfied_<t>`	set instance field
`putfied_<t>_this`	set instance field in associated object
`putfied_<t>_w`	set instance field (using lager index)
`putstatic_<t>`	set static field

Table 3.6: This table contains byte codes which are critical for transactions. They modify fields in the persistent memory which must be restored to their original value if a transaction is approved. `<t>` is a wildcard for a Java Card byte code data type. This includes `byte`, `short` and `reference` (`int` is optional and we don't support it in our implementation).

As all read and write operations to dynamic (and thus also persistent) objects are handled by the *Object Manager* we will intercept write attempts there and record the original values using the transaction mechanism of the OS interface.

3.4.4 Applet Firewall

Requirements

We already introduced the minimal requirements for the Java Card firewall in Section 2.3.4 and explained how applets are separated using a package specific context. In sum, every object gets assigned a context and byte codes can only access objects assigned the same context as the current one.

To allow useful information exchange (and complicate the implementation) several additional rules exist. First there are a number of exception to the general only-same-context rule. The JCRE, as it has a similar role as an OS, can access objects regardless of their context. This is necessary to invoke applets and deliver APDUs (calling `Applet.process`). Arrays and interfaces can be marked as global (calling `JCSystem.makeGlobalArray`) respectively shareable (implementing the interface `javacard.framework.Shareable`), which also means they can be accessed from any context. At last also public methods of the JCRE (called *Java Card RE Entry Point Objects*) can be invoked from any context, as they are indented to be used by all applets.

Another rule which has to be enforced by the firewall regards *Temporary Java Card RE Entry Point Objects*. This are objects, belonging to the context of the JCRE, which cannot be stored in persistent fields (class instance fields or static fields).

Byte Code	**Description**	**Check for**
`getfield`	class instance field access	context
`putfied`	class instance field access	context, temporary entry point obj
`invokevirtual`	method invokation	context, context change
`invokeinterface`	method invokation	context, context change
`throw`	exception handling	context
`<T>aload`	array value access	context
`<T>astore`	array value access	context, temporary entry point obj
`arraylength`	array meta data access	context
`checkcast`	class instance meta data access	context
`instanceof`	class instance meta data access	context
`putstatic`	class static field access	temporary entry point obj

Table 3.7: This table shows byte codes which are critical to enforce firewall rules. Depending on the specific byte code different rules have to be enforced.

Concept

As mentioned before, the foundation of the firewall is the context each object gets assigned. The context is stored in the object struct (field `context` in `obj_s`). The currently active context is assigned to the newly created object (byte code `new` for class instances and `new_array` for arrays). The current active context is stored in the VM frame (`vm_frame_s`) and is usually inherited from the calling method. An exception would be if a method of an shareable interface or if a public method of the JCRE is invoked. In this case the next active context would be the context of the object the invoke is executed on (or the JCRE context which all JCRE classes share).

To enforce the firewall rules stated above, certain byte code have to check if the requested operation is allowed. Table 3.7 shows all relevant byte code and checks which have to be done. Most prevalent is to check if an accessed object has the same context as the currently active one. In the case an object of a different context can be accessed (invoking a method) a context change has to be initiated. The permission to perform a context change is controlled by the flags in the object struct.

Another check that has to be performed is for *temporary entry point objects*. The present of such an object is also indicated by a flag (`temp_epo`). This checks will be performed in the respective byte codes (maybe together with helper functions).

3.5 Software Tools Selection

This chapter will discuss some tools we consider to use in our work. We will separate them in two parts, first we will discuss some tools we will use to implement the specification and design goals. Afterwards we will introduce tools to check the correctness of our implementation.

3.5.1 Development Tools

The tools used for developing have a huge impact on how portable a SW is. The most important 'tool' is the programming language. A programming language, if chosen with care, will do most of the work to abstract the underlying platform. As we work mostly with embedded platforms, it is obvious to use C as the language of choice. To ensure compatibility with different compilers we choose to use a quiet old revision, namely C90[2].

As C is intended for very low level programming some extensions are necessary to use it as a platform independent programming language. Most of the language features will work similar across very different platforms, from 64-bit CPUs with GBs of RAM down to 8-bit CPUs with only a few KB of RAM. This holds for all control flow statements such as if conditions or loops. More difficult are data types as their size can vary across different compiler implementations and also byte ordering (endianes) isn't covered by C.

To overcome this limitations we introduced custom extensions (data types and libraries) to the standard C. They will be supplied through the Common module. They can be largely grouped as followed.

- **Custom Data Types**
  To ensure the same behavior across different platforms (compilers) we will define custom primitive data types for signed (`intX`) and unsigned (`uintX`) numerical values. They are available with 8-bit, 16-bit and 32-bit size. We also define a pointer to a byte field (`ptr`).

- **Custom Library**
  For portability and also memory efficiency we won't link a large library into our project there we only need a few functions. Instead we chosen to implement them in project specific library (`jcos_lib.c`). This library will supply basic functions like copying or comparing memory areas.

- **Keywords**
  Different compiler also use different keywords for example for packed structs. This

[2]Standardized as ANSI X3.159-1989 'Programming Language C.'. As the standard has been withdrawn and replaced with a newer one we used [26] as reference.

keywords will be supplemented with generic expressions in the source code and the final expression will be defined in the common header file.

Some of this extensions will have to be adapted to each platform or compiler. Whereby they are more dependent on the compiler implementation then on the target platform. Multiplatform compiler such as GCC will need less or even no modification. In contrast a different compiler for the same target platform (e.g. x86 running Linux can be targeted with a lot of different compilers) may also need modification.

3.5.2 Test Framwork

To ensure the correctness of our implementation we will provides separate test cases for layers, modules and interfaces. This method is called unit testing. A lot of test frameworks support unit testing for different languages. We decided to use *googletest*, which is a C++ framework, but can integrate C code.

googletest organizes SW tests in test cases. Individual tests can be defined by using the C++ macro `TEST(test_case_name, test_name)`. The first parameter specifies the name of test case and the second parameter the individual test. Inside of such a test two different types of assertions are possible. `ASSERT_*` aborts the current function if it fails, whereby `EXPECT_*` continuous execution even in case of a failure. Both exist in various forms for binary comparison (equal, greater, lower etc.) or to test for boolean values (TRUE,FALSE).

To initialize and reuse a test environment, tests can be assigned to a test fixture (using macro `TEST_F(test_case_name, test_name)`). To initialize the test fixture the test case name has to be the name of a class which inherits from `::testing::Test`. This class has to be defined before the first test and can initialize and cleanup the environment using the class constructor/destructor or implementing the methods `SetUp()` and `TearDown()`. Although this allows the developer to reuse a test environment, the class instance isn't shared across different test. Every time `TEST_F` is invoked a new environment object is created and changes made in tests aren't preserved.

The *googletest* framework invokes all test cases defined by the macros described above and prints the results (see Figure 3.5). In the case of an error additional information is printed. The developer can implement it's own main function or the use one supplied by the framework.

3.6 Summary

In this chapter we presented our approaches for implementing a Java Card VM. Before presenting our ideas for the implementation we described the predefined design goals in Section 3.1.

In the following we described the system architecture in Section 3.2. We proposed a three layer design with lean interfaces between the layers. Only one layer (the HAL) should contain platform specific code, the other two should be HW independent. The well defined interfaces allows to easily exchange a layer.

In Section 3.3 we described the data structs used for representing VM internals. This includes a simple C unit to treat the different VM data types and more complex structs to handle VM frames, methods and dynamic objects (class instances and arrays).

Section 3.4 looks at some more sophisticated features, for which the description don't fit in the previous sections. They are all related to the VM layer but are distributed over multiple modules or layers. This includes the native libraries, the transaction mechanism and the applet firewall.

At last, in Section 3.5, we anticipate a short description of the tools we will use during implementation. As the development tools have a strong influence on the outcome we regarded it necessary to address this topic hear. We presented programming language and the test framework we will use, namely C90 and googletest.

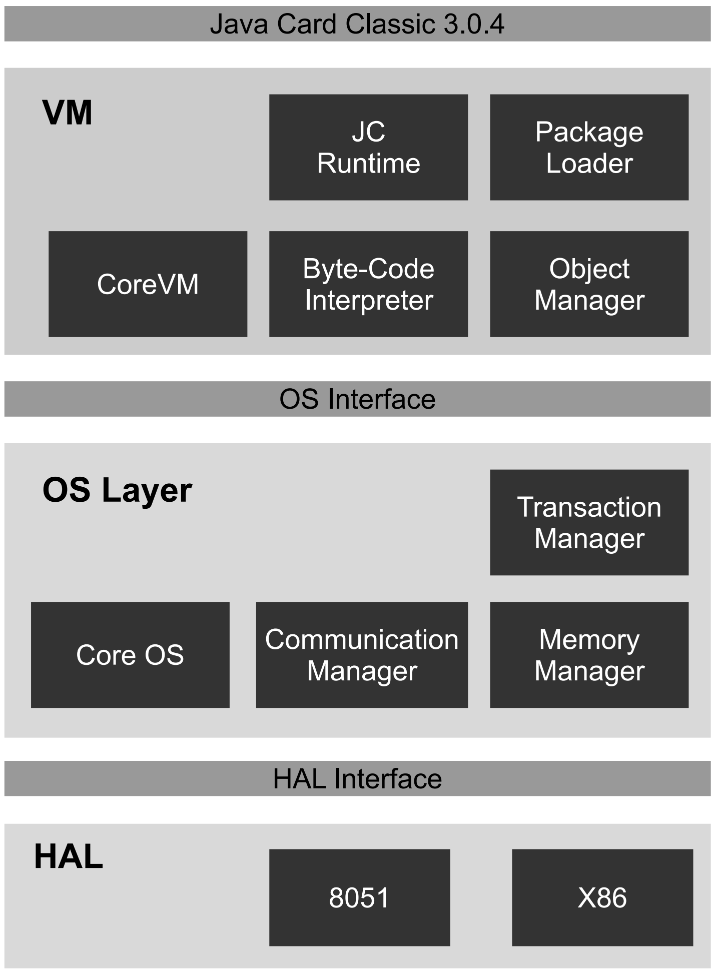


Figure 3.1: This figure shows the layered architecture of the system and the modules inside a layer.

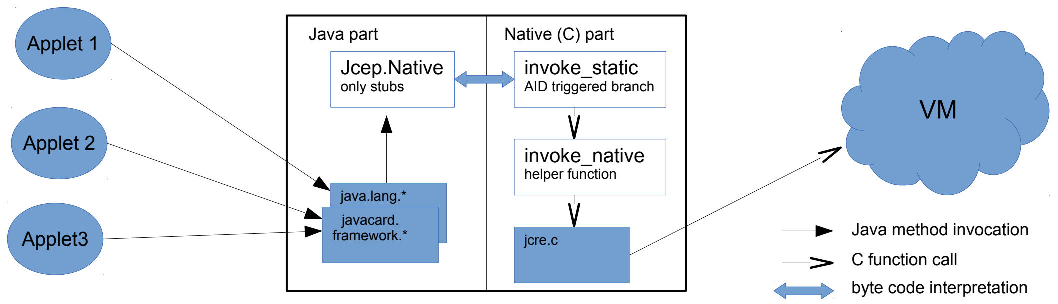


Figure 3.2: This figure shows how library methods are called. Applets can invoke methods in the JCRE Java part in the same way as invoking methods in any package. The JCRE Java part uses native binding to invoke a C functions in `jcre.c`, which can subsequently access all VM resources via C function calls.

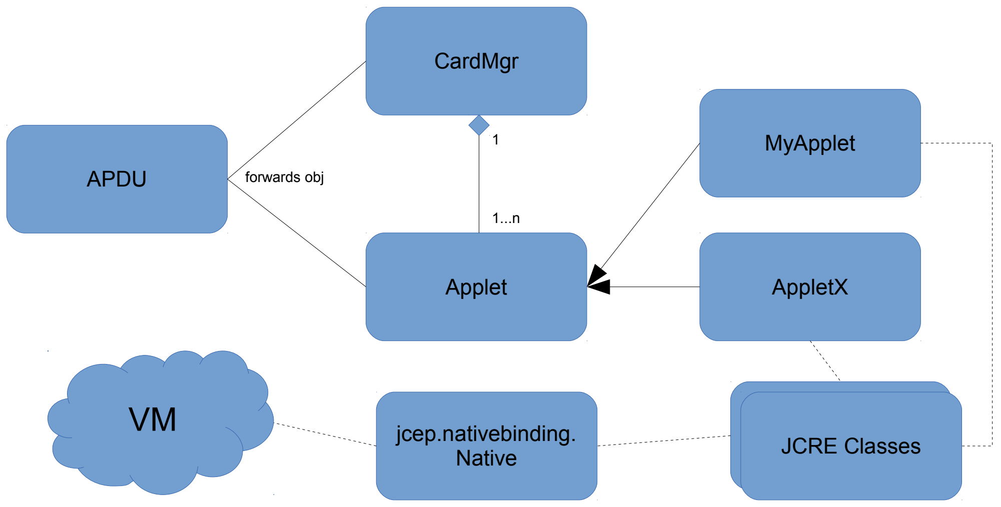


Figure 3.3: This figures shows the Card Manager and a part of the JCRE. The Card Managers maintains a list of all installed applets. Every applet (see examples) must be derived from class `javacard.framework.Applet`. The Card Manager forwards incoming APDUs to the Applets. Applets can use the native functions through the JCRE classes.

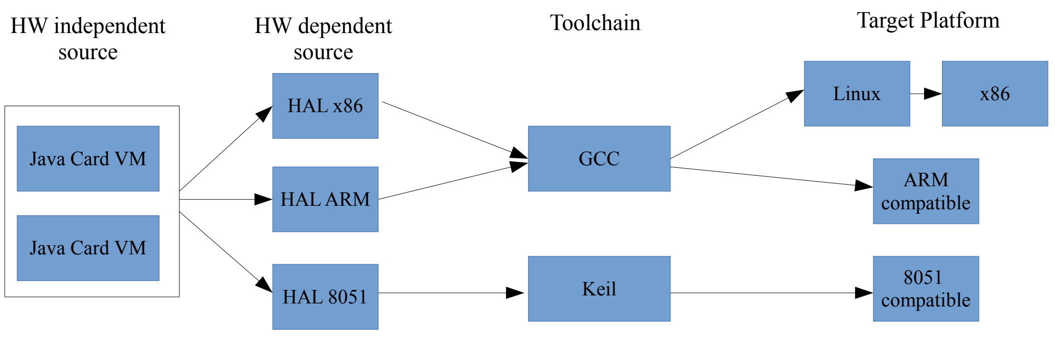
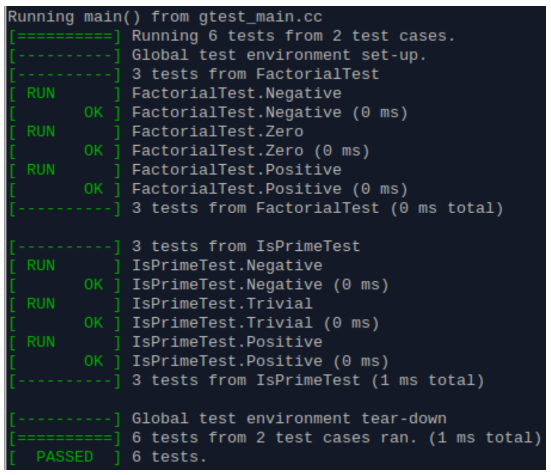


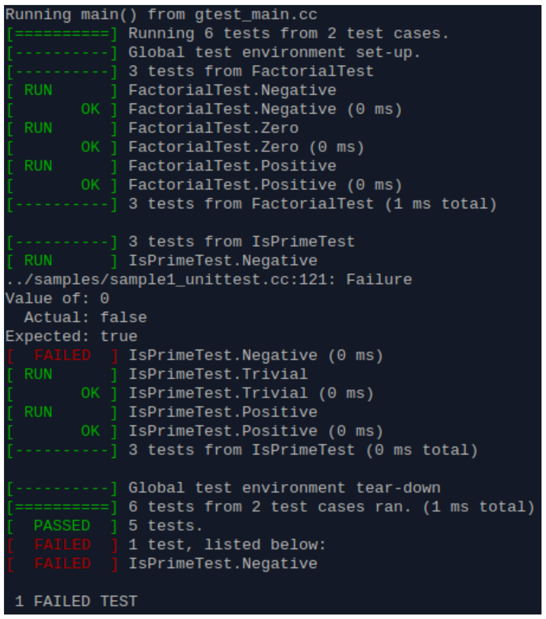
Figure 3.4: This figure shows the toolchain workflow. The hardware independent code (OS Layer and Java Card VM) is supplemented with the hardware dependent code (the specific HAL implementation). This servers as input for the compiler suite. Some compilers (like GCC) support multiple platforms. The generated binary can the be deployed on a HW platform either with or without an OS.



```
Running main() from gtest_main.cc
[==========] Running 6 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from FactorialTest
[ RUN      ] FactorialTest.Negative
[       OK ] FactorialTest.Negative (0 ms)
[ RUN      ] FactorialTest.Zero
[       OK ] FactorialTest.Zero (0 ms)
[ RUN      ] FactorialTest.Positive
[       OK ] FactorialTest.Positive (0 ms)
[----------] 3 tests from FactorialTest (0 ms total)

[----------] 3 tests from IsPrimeTest
[ RUN      ] IsPrimeTest.Negative
[       OK ] IsPrimeTest.Negative (0 ms)
[ RUN      ] IsPrimeTest.Trivial
[       OK ] IsPrimeTest.Trivial (0 ms)
[ RUN      ] IsPrimeTest.Positive
[       OK ] IsPrimeTest.Positive (0 ms)
[----------] 3 tests from IsPrimeTest (1 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 2 test cases ran. (1 ms total)
[  PASSED  ] 6 tests.
```



```
Running main() from gtest_main.cc
[==========] Running 6 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from FactorialTest
[ RUN      ] FactorialTest.Negative
[       OK ] FactorialTest.Negative (0 ms)
[ RUN      ] FactorialTest.Zero
[       OK ] FactorialTest.Zero (0 ms)
[ RUN      ] FactorialTest.Positive
[       OK ] FactorialTest.Positive (0 ms)
[----------] 3 tests from FactorialTest (1 ms total)

[----------] 3 tests from IsPrimeTest
[ RUN      ] IsPrimeTest.Negative
../samples/sample1_unittest.cc:121: Failure
Value of: 0
  Actual: false
Expected: true
[  FAILED  ] IsPrimeTest.Negative (0 ms)
[ RUN      ] IsPrimeTest.Trivial
[       OK ] IsPrimeTest.Trivial (0 ms)
[ RUN      ] IsPrimeTest.Positive
[       OK ] IsPrimeTest.Positive (0 ms)
[----------] 3 tests from IsPrimeTest (0 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 2 test cases ran. (1 ms total)
[  PASSED  ] 5 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] IsPrimeTest.Negative

 1 FAILED TEST
```

Figure 3.5: This figure shows a sample output from *googletest*. The output on the right sight shows a test run with all tests succeeded. On the left side the side is the output with an error. *googletest* supplies additional information like the name of the test case and the line number.

Chapter 4

Implementation

This section will describe the major aspects occurred during the implementation. First we will describe the tools we used and why we used them. Afterwards we will show how some use cases are handled. At the end we will describe the test cases we will use to show the correctness of our implementation.

4.1 Development Tools

In this section we will describe the tools we used for our implementation. We already gave an outlook to the development tool we would use in Section 3.5. Now we will extend the description with additional tools we selected during the implementation process.

4.1.1 Eclipse

Eclipse started as a closed source IDE developed by IBM. Today it's a platform for a broad range of software products. This includes IDEs for many different languages including the Java Development Tools (JDT) and the C/C++ Development Tools (CDT). Eclipse also provides a foundation for non-IDE applications, called the *Rich Client Platform*. The wide field of applications is made possible by an architecture where nearly everything is a plugin, even the JDT is not privileged in comparison to other programming language plugins. [27]

The development of Eclipse is overseen by the Eclipse Foundation. It is a non-profit organization which is funded by its members (include companies like Google, Novell, IBM and many more) and led by a Board of Directors. [28]

We used Eclipse together with the CDT throughout the implementation and the JDT for parts of the JCRE. We chose Eclipse because of its code editor (includes live code competletition and checking) in general and its good support for C.

4.1.2 GCC

The GNU Compiler Collection (GCC) is a set of compilers for multiple programming languages including C/C++, Objective-C and ADA. GCC can produce machine code for different processors. Language dependent parts are called front-ends while processor dependent parts are called back-ends. [29]

The prevalence of the GCC on a wide array of target platforms and for different input languages let us chose it as the main compiler for development. The C front-end also provides a strict mode (compiler argument `-std=c89`) to enforce the use ANSI C compatible (see Section 3.5) input code.

4.1.3 Netbeans

Netbeans is a Java based IDE and supports development in different programming languages. Among them are most notable Java but also PHP and C/C++. It can be enhanced with plugins, for example to build Java Card applets directly within the IDE. Altogether Netbeans is comparable to Eclipse. [30]

We used Netbeans, in addition to Eclipse, because of the better support for Java Card development. We didn't found a comparable Java Card development plugin for Eclipse.

4.1.4 Java Card Development Kit

The JCDK is provided by Oracle for developing Java Card applets. It contains a converter tools which transforms Java class files into CAP files and a simulator to test these applets. The export files (see Section 2.3.3) of the JCRE classes are also included and can be used to implement a custom but compatible JCRE. [31]

We used the JCDK to convert our JCRE classes together with the supplied export files and to convert our test applets.

4.1.5 codavaj

codavaj is freely available (under Apache License V2.0) reverse engineering tool to convert JavaDoc into Java source files including class definitions and method stubs. It supports Java 6 including Generic and Annotations. [32]

We used codavaj to generate class definitions and method stubs for the JCRE classes. This not only saves as time for rewriting the class definitions by hand but also eliminates transcriptions errors.

4.1.6 Keil

Keil is a set of SW tools for developing embedded systems applications. It include the IDE uVision, compilers for different processors and a debugger/simulator. The compiler for 8051 processors is termed C51. [33]

We used Keil uVision to port our implementation to a 8051 compatible platform.

4.2 Test and Analysis Tools

In this section we will describe tools we used to verify the functional and non-functional requirements.

4.2.1 googletest

googletest is a testing framework based on xUnit (which is derived from JUnit). We already described it's capabilities in Section 3.5. [34]

We used googletest to for unit testing of our implementation. This should help us to avoid general programming errors. It will be of limited use for integration tests on different platforms as it's not intended for embedded systems (googletest needs a C++ compiler which is not always available).

4.2.2 Cyclomatic Complexity Analyzer

This tool can be used, as the name already implies, to calculate the Cyclomatic Complexity (CC) for a complete source code directory. Additionally it counts the Lines of Code (LOC) per function. The obtained information can be exported to a CSV file.

We used the Cyclomatic Complexity Analyzer for the analysis of our source code regarding the complexity and volume. This values can be used to validate some of our design goals.

4.3 Use Cases

The following section will describe some major use cases and operations of our implementation. In contrast to Chapter 3, which focused on static properties like modules and data types, this section will deal more with dynamic data and execution flows.

4.3.1 VM Initialization and Startup

This use case will describe the steps needed to launch the byte code execution and to invoke a main function. At the beginning the runtime data area for the VM has to be initialized. We decided to organize this data in two stacks, one for the VM frames and another one for local variables and operands.

For each invoked method, a new frame is pushed on the first stack. The frame contains all meta data of the current executed method and references to the second stack. For a detailed description see Section 3.3.2.

The second one contains the local variables, including the arguments of a method, and the operands. Arguments are treated like local variables and are placed before the operands. This enables handing over arguments without copying them to another memory locations. Instead, in the new VM frame, the reference to the local variables is set to the location in the stack where the first argument was placed. The invoked method is allowed to change them because all local variables are removed from the value stack when a method returns. The structure of the two stacks and links between them can be seen in Figure 4.1.

To start up the VM, we first initialize the above described stacks by calling the function `init_VM_stack`. Both stacks get assigned a fixed size area in the transient part of the memory. The maximum stack size can be adjusted to specific platforms using the two constants `MAX_VM_FRAMES` and `MAX_VALUE_STACK`. The VM stack is further initialized by pushing a frame on it which references the base of the value stack.

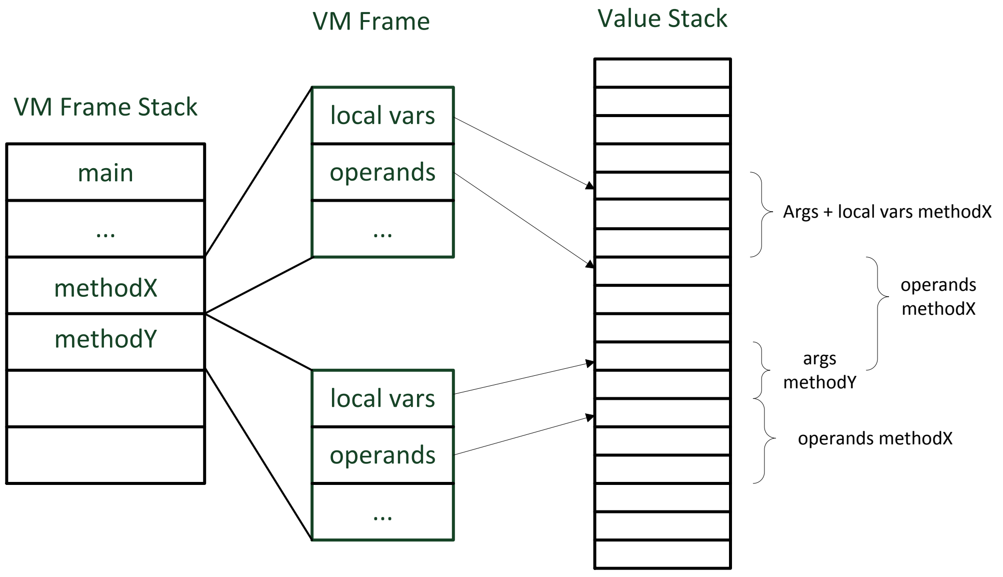


Figure 4.1: This figure shows how the runtime data is organized. The VM frames reference the value stack, on which local variables and operands are stored interleaved. The operand area of the calling method and the local variable area of a called method overleap enabling the called method to access arguments, pushed on the stack before right before, without copying them to a new position.

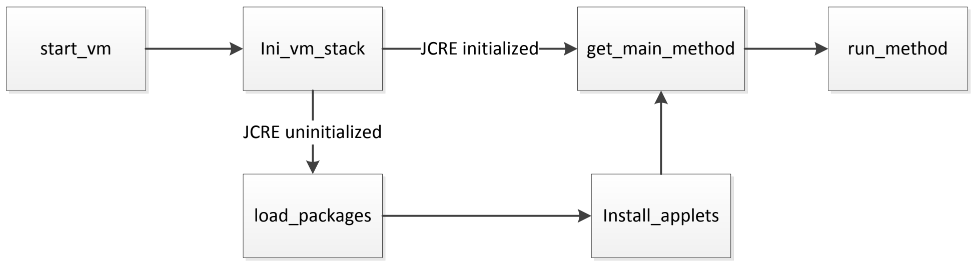


Figure 4.2: This figure shows the activities needed to start byte code execution. Names in the rectangle are the appropriate C functions.

If not done in a previous session, packages are installed using the function `load_packages`. Present packages are added to a global list (variable `installed_packages` of type `struct cap_container_s`), which is serving as a container for all packages available on the system. If a package define one or more applet (indicated by having a *Applet Component*), `install_applet` is called. This function retrieves the install method and executes it using the functions `load_method` and `run_vm` with the currently installed package as context.

The first method to be executed is always `CardManager.main`. It is a static method with no arguments located in the package of the `Card Manager`. Public static methods can be retrieved using the class and method token and the export component. The tokens for the main method are supplied through the constants `CLASS_TOKEN_CARDMGR` and `METHOD_TOKEN_CARDMGR`. The function `get_main_method` performs a lookup the CAP file of the Card Manager. The package is accessed through its index supplied by the constant `CAP_INDEX_CARD_MGR`. In the final step the main method is executed in the same way as the install methods but using JCRE context. Figure 4.2 shows the particular steps needed to to startup the VM.

4.3.2 Method Retrieval and Invocation

Since Java provides object oriented programming on byte code level, including inheritance and overriding, the invocation of a method invocation can request different amounts of lookups. Java Card provides different byte codes for different kinds of invocation (see Table 4.1). Depending on the used byte code and the requested method, an invocation can lead to multiple runtime checks and lookups.

Regardless of the type of invocations, once the meta data and the byte code of a method are retrieved the same process to execute a method can be used. All arguments, including the this reference, are pushed on the stack. In the following the byte code interpreters calls the function `load_method` and supplies all retrieved data as a `method_s` struct (see Section 3.3.3). Inside this function a new VM frame is created (see Section 3.3.2) and the dispatch loop continues with the first byte code of the invoked method.

Byte Code	**Method Properties**	**Possible Lookups**
`invokestatic`	static	external package
`invokevirtual`	non-static, non-private	external package, implementing class
`invokespecial`	private or super	external package, implementing class
`invokeinterface`	defined in an interface	external package, implementing interface + class

Table 4.1: This table shows the different byte code for invoking a method. Depending on the byte code different lookups may have to be performed.

In the following we will look at the different cases. Starting with the most simplest one, the invocation of a static method, up to the most complex case, the lookup of an interface method. Figure 4.3 shows an outline of how different invoke commands are handled.

Static and Private Methods

Methods which are declared static or private can be retrieved with the least effort as they can be identified at compile time. The Java compiler and the Java Card converter supply the exact position of the method byte code in form of an offset into the method component. The *Package Loader* doesn't need to perform any lookup in the class hierarchy. They method may be implemented in a different package (stored in a different CAP file). If this case happens, the *Package Loader* has to perform a lookup for the requested package given its AID.

For invoking a private method a class instance has to be supplied and is handed over to the method as first argument. Static methods can be invoked without an class instance.

Virtual Methods

Virtual methods are usually the most prevalent type in Java since every method which is not assigned any special attribute is a virtual method. Virtual methods can be overridden multiple times in sub classes. Since the type of a reference and the type of the referred object may not be the same, the runtime has to perform a lookup for the requested method. For this purpose the class hierarchy is traversed upwards starting at the object type. If matching method is found in the object type or in any of its superclass, this method will be used. If no overriding method is found, at least the call of the reference type must implement the requested method.

Due to the more complex method retrieval, developers of Java Card applets are encouraged to use static methods when possible.

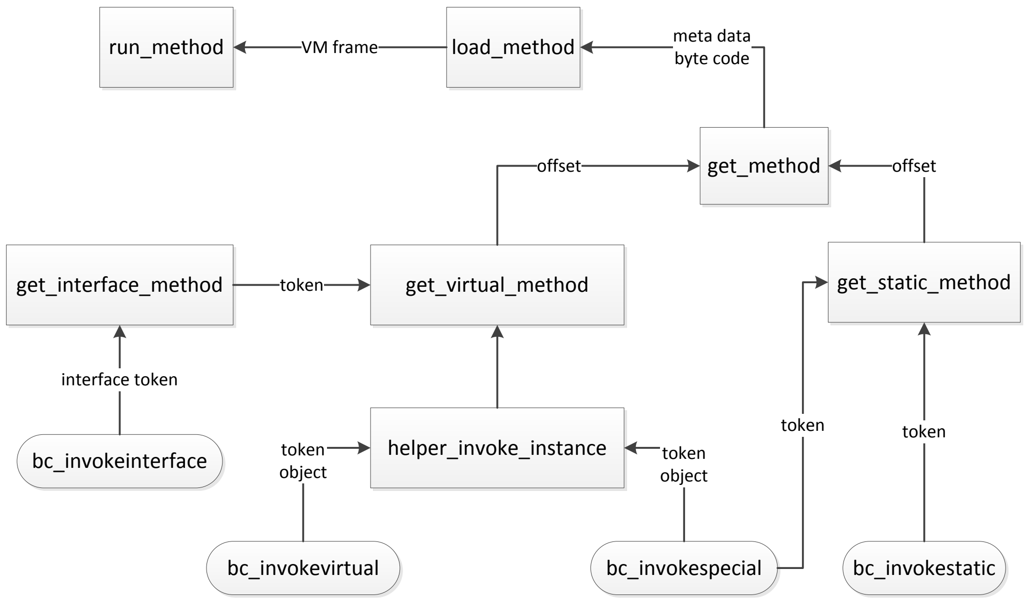


Figure 4.3: This figure show the processing of different invocation byte codes. Arrows indicate information flow (not control). Tokens denote logical data (e.g. index in table), whereby offset stands for a physical address inside a CAP.

Interface Methods

References can also be of type interface. Interfaces are pure abstract classes which do not implement any methods. In Java, classes can implement multiple interfaces which supports some kind of multiple inheritance, normally forbidden in Java. This feature makes the retrieval of an interface method quite complex. First the implementing interface has to be determined by traversing the class hierarchy upwards and looking for a class implementing the interface of the reference type. Given a class hierarchy with one or more superclasses, the same interfaces can be implemented by more than one class. In this case, subclasses override interface methods being already implemented in a superclass. When the implementing class is found, the interface method can be matched to a virtual method. It may be necessary to perform a lookup for the virtual method in the way we already described above.

Due the large overhead when invoking an interface method, this should be avoided by developers whenever possible.

4.3.3 Applet Selection and APDU Processing

Java Card applets communicate with the external world using APDUs. We already explained them in Section 2.3.1. In the following we will show how APDUs are processed through the layers and in the VM. We will also show how APDUs can be used to select an applet.

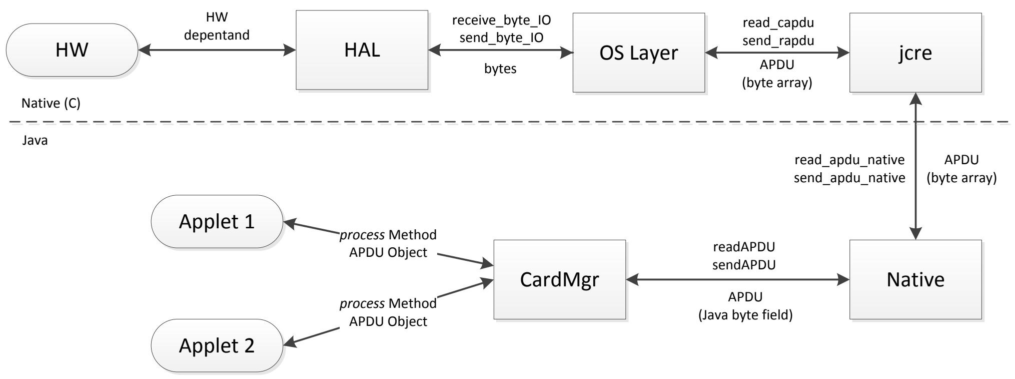


Figure 4.4: This figure show the steps how an APDU is processed, starting at the underlying HW, through the HAL and the OS layer, up to the VM. The incoming bytes are assembled to an APDU, stored in a byte array and included in a Java object.

Forwarding APDUs

In the lowest layer, the *HAL*, APDUs are received as single bytes. This can be done in quiet different ways depending on the underlying HW. Some platforms for example may provide library functions like *getchar* or similar ones. On other platforms the I/O is maybe memory mapped or has special CPU instructions. The same assumptions hold for outgoing data.

Regardless of how the I/O data processed by the HW, the HAL will offer functions to send and receive single bytes. At this point there is no cohesion between the processed bytes. The first interpretation of I/O data is done by the *OS layer*. In this stage, the individual received bytes are arranged to an APDU. The other way around, when sending an APDU, would be to disassemble an APDU and transmit it byte per byte.

The OS layer is expected to wait until a full APDU was received (or was sent). When the complete APDU is placed in a byte array we specified before, the control is handed back to the VM. The mentioned byte array is the same used as buffer in the APDU object (class `javacard.framework.APDU`), so the APDU is already stored in a memory area which we can be accessed by Java code. All processing steps of an APDU, from HW to Java byte code, are shown in Figure 4.4.

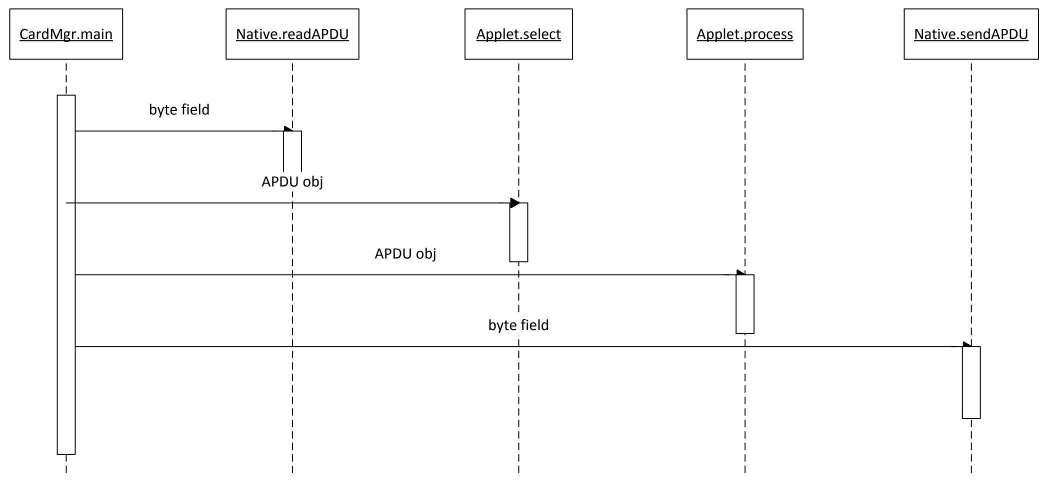


Figure 4.5: This figure show the call sequence for selecting an applet.

Selecting an Applet

When receiving an incoming APDU, the JCRE forwards it to the currently selected applet. After starting up a Java Card device, the selected applet is always the *Card Manager* (also called default applet). The CAD can select another installed applet by sending a special APDU, described in Table 4.2.

Field	**Value**
`CLA`	0x00
`INS`	0xA4
`P1`	0x04
`P2`	0x00
`LC`	AID length
`Data`	AID

Table 4.2: The table shows the content of an APDU thought to select an applet.

The task of applet lookup and selection is done by the Card Manager. Every incoming APDU is checked if it contains an applet selection. In this case the Card Manager invokes `Applet.deselect` on the currently selected applet. In the following the received AID is checked against all installed applets and if a fitting applet is found, `Applet.select` is invoked on it. Anyway the received APDU is forwarded to the selected applet by invoking `Applet.process`. Figure 4.5 shows the procedure of selecting an applet and forwarding an APDU.

4.4 Test Cases

The following section will describe the test cases we used to verify the correctness (and compliance) of our implementation. We will use layer tests, which tests the functionality of an entire layer, and module tests. The tests are constructive, for example we will test the *OS Layer* with an already tested implementation of the *HAL*. Module tests will also be constructive to avoid simulating behavior of all other modules when testing one.

4.4.1 HAL Test

Functional testing of the HAL is difficult because the HAL is tailored for each platform. Due to this circumstances, meaningful testing of a HAL implementation can only take place on the target platform the implementation was designed for. To be able to test on a wide variety of target platforms we foresee the use of a testing framework which needs a C++ compiler. Instead we provide a test module in a single C source file (`hal_test.c`) which tests the two basic functions of the HAL: memory allocation and communication.

The test case includes the following tasks:

- **Startup and Hello World Message**
  The test module is started by the HAL and tries to message through the outgoing channel. If this message is not printed, the HAL was not able to boot or to launch any further execution.

- **Communication Test**
  The test module examines the ability of the HAL to communicate through incoming and outgoing channels. Characters received through the incoming channel are forwarded (echoed) to the outgoing channel without any modification. This should prove the ability to receive and send data correctly. The input and the output data can be compared outside the HAL under test to ensure a faultless communication.

- **Memory Access Test**
  The test module tries to access the entire memory the HAL grants him and fills them with number according to different patterns. The transient memory is byte wise filled with increasing numbers starting from 0 to 255 and the persistent memory with decreasing numbers from 255 to 0. When the range is exceeded counting starts at the initial value. After both memory areas are filled with numbers according to a different pattern they are read again and checked for errors. If a number was not written or was written in the wrong memory area the patterns support detection of the error.

To run the HAL test, the project has to be built with the preprocessor flag `TEST_HAL`. When this flag is present the HAL implementation should start `run_hal_layer_test` instead of `start_os`. To ensure correct handling by the HAL implementation the definition of `start_os` is removed from the *HAL interface* when the flag is present.

4.4.2 OS Layer Test

This test case should verify the correctness of the *OS Layer*. Due to the OS Layer doesn't access platform functions directly, it can be tested independently from the target platform.

The OS Layer will use the HAL implementation for X86, which will be tested before using the test case stated in the previous section. To organize the tasks of this test case we will use *googletest* which we already introduced in Section 3.5.2.

The test case includes the following sub tests:

- **Memory Management**
  This test will check the ability of the OS Layer to organize the memory. First memory areas with different size are reserved, using the function `malloc` of the *OS Inteface*, until the memory has exceeded. The memory areas will be filled with the same pattern used in the HAL test: increasing numbers in each byte, wraping around when number space exceeds. In the next step the patterns are checked for errors which will occur for example if reserved memory areas overlap. In the following all previous reserved memory areas are freed and tried to reserve again.

- **Transaction Management**
  This test will check the ability of the OS Layer to undo modification recorded during a transaction. For this purpose several memory areas are reserved using the `malloc` function of the OS Interface. The memory areas are initialized with specific values for each area. In the following a transaction is started by labeling some areas as part of the transaction (using function `record_transaction`). Modification of these

areas should be reverted when calling `rollback_transaction` or made permanent when calling `commit_transaction`. Both cases are verified by the test environment.

- **Communication**
  This test will check the ability of the OS Layer to correctly parse an incoming APDU and to disassemble an outgoing APDU and transmit it byte wise. The incoming APDU is provided by the test environment and transmitted using the x86 HAL implementation. The test environment will receive the APDU through the OS Interface and echoes it through the same (only the data). The test environment will compare both to check if any errors occurred during transmission.

Due to the fact the used test environment needs a C++ compiler, the OS Layer and the HAL are compiled separately for this test case. The tasks of this test case are implemented in `os_layer_test.cc`. The test environment doesn't need to declare a main function because *googletest* is able to find all test cases through the used macros.

4.4.3 Package Loader Test

In contrast to the layer tests described above isolation of the VM modules is not that easy. The unit tests described bellow sometimes have to use functionality of other modules which aren't explicitly targeted in that test case. In this case we will try only to use functionality which was checked before by another test case or we will provide mockups.

- **Constant Pool Access**
  This test will perform a lookup in the Constant Pool Component of the first package and tests if the item can be parsed correctly. The item is specified by its index. The function under test is `get_constant_pool_info`.

- **Method Lookup by Offset**
  This test will verify if a method can be parsed correctly into a `method_s` struct (see Section 3.3.3). The method is specified by the offset and the containing package. The function under test is `get_method`.

- **Static Method Retrieval**
  This test will check the ability of the module to lookup a static method. A static method is specified by a Constant Pool Entry. It may requests a lookup for an external package and parsing a method. The function under test is `get_static_method`.

- **Virtual Method Retrieval**
  This test will check the retrieval of a virtual method. A virtual method can be identified by the reference type, the object type and the method token. The test

environment will provide a class instance struct which represents the Java object. This test case involves a lookup in the class hierarchy of the object and a method lookup. The function under test is `get_virtual_method`.

- **Interface Method Retrieval**
  This test will check the retrieval of an interface method. The setup is similar to test for virtual method retrieval expect the reference type is an interface. Through that the test case also involves the lookup for the implementing interface. The result should be again a parsed method. The function under test is `get_interface_method`.

The tests are implemented in `package_loader_test.cc` and are again organized by *googletest*. The packages used for testing is `jcos.testing` and `jcos.testing.extern`.

4.4.4 Object Manager Modul Test

Due to the *Object Manager* uses functions of the *Package Loader Test* this test case has to be performed after the testing the latter module. When the former test is finished we can rely on correct behavior of the *Package Loader* module. The *Object Manager* performs various operation on dynamic objects, like creating and accessing arrays or class instances.

The test case includes the following sub tests:

- **Array Creation and Access**
  This test will perform the creation of Java arrays of different types. In the following array access (put field and get field) will be tested. Due to object deletion is optional and we do not support it, we also will not test it. The functions under test are `array_new`, `array_load` and `array_store`.

- **Class Creation**
  This test will check the creation of a derived class, which includes the creation of instances for all super classes. The resulting class also serves as input for the next test. The function under test is `new_obj`.

- **Field Access**
  This test will check the field access for the previous created class. This will include access to fields of the super classes. The functions under test are `get_static_field` and `put_static_field`.

- **Type Checking**
  This test will check the ability of the *Object Manager* to perform type checking in dynamic types. This functionality is needed to perform save casts in Java and for the `instanceof` operator. The function under test is `check_type`.

The tests will be performed on special test classes implemented in the package `jcos .testing`. One class will have at least two super classes (including `Object`) and will implement two interfaces at different stages in the class hierarchy.

4.4.5 Byte Code Module Test

The *Byte Code Module* has a vast number of functions but not all require automated testing. A lot of the byte codes are arithmetic operations or simple stack manipulation. Another large set of byte codes heavily uses the functionality of the other modules. All these byte codes (and thus functions) have in common to contain very little code. In our opinion, this condition reduces the need for automated testing as short code segments can be easily checked by the programmer.

Nevertheless some function contain significant amount of code and are included in the automated tests. They can be grouped as follows:

- **Branch Statements**
  The Java Card byte code knows many different commands to control the program execution. The available options range from an unconditioned goto command, over jumps depending on various conditions to complex byte codes which reassemble switch-case commands on byte code level. For this test we will check the program counter after executing a branch control command.

- **Invoke**
  Although most of the functionality to invoke methods is implemented in the *Package Loader*, functions implementing the different invoke byte codes contain significant amount of code. To ensure the correct behavior we will test the byte code functions together with the *Package Loader* at this place.

To execute the above described tests, we will build up on the previously tested modules. In addition the test environment will provide a valid VM frame for execution (see description in Section 3.3.2).

4.4.6 JCRE Test

This test case will cover functions of the *JCRE Module* which can be easily isolated. This holds for most of the Java only part and for the implementation of the native methods. The collaboration of the Java and the native part will be covered in the *System Integration Test*.

The tested functions/methods can be grouped as followed:

- **Card Manager**
  The Card Manager is implemented in Java and accesses native functions through `jcos.Native`. This test only covers the Java part which involves methods for registering and retrieving applets.

- **Native Functions**
  These functions serve as the native backend for the Java method stubs in `jcos.nativebinding.Native`. Their purpose range from simple manipulation of native variables over communication handling to memory management. The test will cover only functions with significant amount of code and will utilize the googletest framework.

- **Applet Installation**
  Installing applets involves a lookup in the associated CAP file and invoking a Java method from a native context. These two tasks are essential capabilities which makes them worth testing. The test environment will provide a package (CAP file) containing an applet. In the course of the installation process the method `CardMgr.addApplet` should be invoked. In this case the test has succeeded.

4.4.7 System Integration Test

This test case will examine the *Core Module* of the VM together with the other modules and the underlying layers. We will use the reference implementation as the golden device. The test environment will compare the response APDUs created by our implementation with the ones created by reference implementation.

The individual tests will be some of the sample applets, which are contained in the JCDK, together with a custom test applet.

- **JCDK Hello World**
  This applet from the JCDK demonstrates the ability of our implementation to work with APDUs (and also to execute byte code in general). The data of the command APDU is read and echoed through a response APDU. The test succeeds then the received data is identical to the data sent.

- **JCDK Wallet**
  This applet from the JCDK represents an electronic wallet. The supported operations are to credit or debit the balance and to protect the wallet using a PIN. The results will be checked against the results of the reference implementation.

- **Calculator Applet**
  This applet is part of our JCRE implementation and provides a Java Card bases calculator. The applet is supplied with two operands and an instruction value to select a mathematical function. The applet can calculate addition (`INS=0`), subtraction (`INS=1`), power (`INS=2`) and factorial (`INS=3`). To demonstrate the object-oriented features of the Java Card platform the calculator uses the *command pattern*. The different calculations are all done by classes implementing the interface `ICalc`. The `process` method selects a specific implementation and invoke the method `calc`, which is defined in the interface.

The test applets will be compiled and converted, using the tools supplied with the JDK and the JCDK, prior to running the test case. The test environment will contain the byte code, command APDU and the reference response APDU to check for correct execution.

4.5 Summary

This chapter dealt with the implementation of the concept proposed in Chapter 3. First, in Section 4.1, we described the tools we used for the implementation and explained for what purposed we used them.

In the following, in Section 4.3, we demonstrated the dynamic behavior of our implementation. We showed the major use cases and elaborated how they are processed. This description of the dynamic behavior of our implementation should act as a supplement to the description of the static architecture in Chapter 3.

Finally, in Section 4.4, we explained how to test our implementation. An important aspect is, that test cases are constructive. To test the OS Layer we need a working HAL implementation. To test the Byte Code Module, we need a woking Package Loader, a working Object Manager module and working implementations of all underlying layers. We will conform to this demand by testing the layers/module in the stated order. The results of the test cases will be discussed in Chapter 5.

Chapter 5

Results and Evaluation

This chapter will present the results obtained during testing our implementation. First we will look at tests of the functional requirements derived from the design goal **Compliance** (see Section 3.1). The test cases were already described in Section 4.4. In the following, we will look at the nonfunctional requirements, consisting of the design goals **Portability** and **Maintainability**.

5.1 Functional Requirements

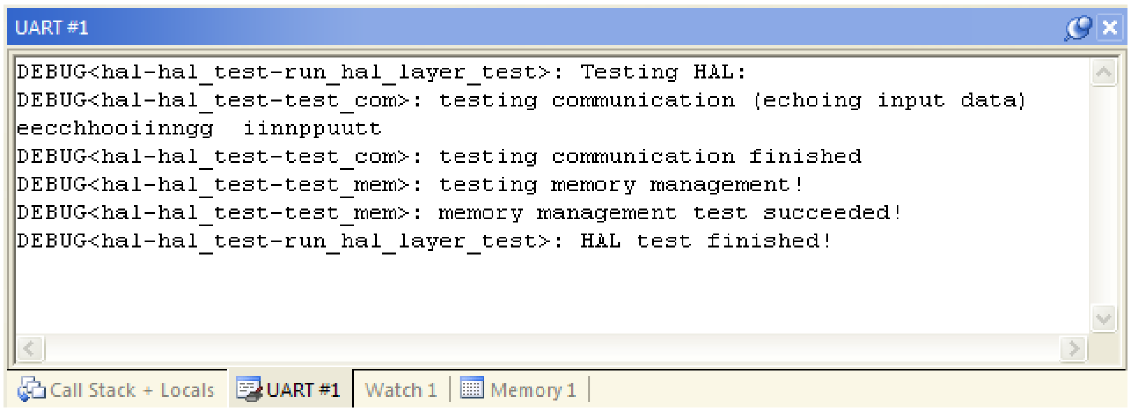
This section will show the test result derived from functional requirements of our implementation. The subsections are organized in the same manner as in Section 4.4. For each test case described before, there will be a subsection in this section containing the results.

5.1.1 HAL Test

Below are the results of the test case we described in Section 4.4.1. The tests were executed using a x86 executeable compiled with GCC and a 8051 executebale using the Keil simulator. When running the test cases we linked C std lib to give more debug information by using printf (preprocessor flag `DEBUG`).

```
<DEBUG>Starting JCEP on x86!
<DEBUG>HAL Test
DEBUG<hal-hal_test-run_hal_layer_test>: Testing HAL:
DEBUG<hal-hal_test-test_com>: testing communication (echoing input data)
0, a4, 4, 0, a, a0, 0, 0, 0, 62, 3, 1, c, 1, 1, 7f,
DEBUG<hal-hal_test-test_com>: block termination
80, 10, 1, 2, a, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, 7f,
DEBUG<hal-hal_test-test_com>: block termination
```

```
DEBUG<hal-hal_test-test_com>: testing communication finished
DEBUG<hal-hal_test-test_mem>: testing memory management!
DEBUG<hal-hal_test-test_mem>: memory management test succeeded!
DEBUG<hal-hal_test-run_hal_layer_test>: HAL test finished!
<DEBUG>Shutting down x86 HAL!
```



```
DEBUG<hal-hal_test-run_hal_layer_test>: Testing HAL:
DEBUG<hal-hal_test-test_com>: testing communication (echoing input data)
eecchhooiinngg  iinnppuutt
DEBUG<hal-hal_test-test_com>: testing communication finished
DEBUG<hal-hal_test-test_mem>: testing memory management!
DEBUG<hal-hal_test-test_mem>: memory management test succeeded!
DEBUG<hal-hal_test-run_hal_layer_test>: HAL test finished!
```

Figure 5.1: This figure show output of the HAL test running on Keil. The input `echoing input` is echoed

5.1.2 OS Layer Test

This test case is described in Section 4.4.2 and is executed using *googletest*. The source code is located in `os_layer_test.cc`. The test case is named `OS_LAYER_TEST`. The result can be seen in the *googletest* output.

5.1.3 CAP Access Modul Test

This test case is described in Section 4.4.3 and is executed using *googletest*. The source code is located in `os_layer_test.cc`. The test case is named `CAP_ACCESS_TEST`. The result can be seen in the *googletest* output.

5.1.4 Object Manager Modul Test

This test case is described in Section 4.4.4 and is executed using *googletest*. The source code is located in `os_layer_test.cc`. The test case is named `OS_LAYER_TEST`. The result can be seen in the *googletest* output.

5.1.5 Byte Code Modul Test

This test case is described in Section 4.4.5 and is executed using *googletest*. The source code is located in `os_layer_test.cc`. The test case is named `OS_LAYER_TEST`. The result can be seen in the *googletest* output.

5.1.6 JCRE Test

This test case is described in Section 4.4.6. The native part is tested using *googletest*. The source code is located in `os_layer_test.cc`. The test case is named `OS_LAYER_TEST`. The result can be seen in the *googletest* output.

5.1.7 System Integration Test

In the following we will show the results of the test cases described in Section 4.4.7. For each test case we show the input (command APDU) and the output data (response APDU). All applets are selected by a special APDU described in Section 4.3.3. This APDU will also be the first input. We added line number to ease reading.

JCDK Hello World

The following communications was generated by the *Hello World* applet running on our Java Card Operating System (JCOS). The input consists of three APDUs. The first APDU (imput line 1) will bring the Card Manager to select the applet but is ignored by the applet itself. The result is the status word for normal execution (`90 00` output line 5). Afterwards the applet receives two APDUs (input line 2 and 3) which are echoed and trailed with a status word (output line 6 and 7).

Input (APDUs):

```
1: 0x00 0xa4 0x04 0x00 0x0a 0xa0 0x00 0x00 0x00 0x62
   0x03 0x01 0xc 0x01 0x01 0x7F;
2: 0x80 0x10 0x01 0x02 0x0f 0x01 0x02 0x03 0x04 0x05
   0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
   0x7F;
3: 0x80 0x10 0x01 0x02 0x00 0x7F;
```

Output (APDUs + debug messages):

```
1: <DEBUG>Starting JCEP on x86!
2: <DEBUG>Starting CardOS
3: <DEBUG>VM not previously initialized, 6 packages loaded
```

```
4: <DEBUG>Start VM (card manager)
5: 0x90 0x0 ;
6: 0x80 0x10 0x1 0x2 0xf 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa
   0xb 0xc 0xd 0xe 0xf 0x7f 0x90 0x0 ;
7: 0x80 0x10 0x1 0x2 0x0 0x7f 0x90 0x0 ;
8: <DEBUG>No further input data (APDUs), shutting down JCOS!
```

JCDK Wallet

The *Wallet* applet is bundled with the JCDK and simulates the behavior of a simple electronic wallet. The four provided functions are: verify a PIN (input line 2), retrieve the balance (input line 3), credit the balance (input line 5) and debit the balance (input line 6). The input (command) APDUs generates a respond APDU which consists of a data segment containing the balance (eg. output line 3 and 6) and a status word. If the applet doesn't send any data only the status word is returned. The status word is `0x90 0x0` if no error has occurred (stands for `SW_NO_ERROR`) or represents and error code (e.g. output line 4: `0x6a 0x85` stands for `SW_NEGATIVE_BALANCE`).

A part of the input APDU script supplied by the JCDK, the full script can be found in Appendix D.1):

```
2: 0x80 0x20 0x00 0x00 0x05 0x01 0x02 0x03 0x04 0x05 0x7F;
3: 0x80 0x50 0x00 0x00 0x00 0x02;
5: 0x80 0x30 0x00 0x00 0x01 0x64 0x7F;
7: 0x80 0x40 0x00 0x00 0x01 0x32 0x7F;
```

Output (APDUs + debug messages):

```
<DEBUG>Starting JCEP on x86!
<DEBUG>Starting CardOS
<DEBUG>VM not previously initialized, 9 packages loaded
<DEBUG>Start VM (card manager)
1: 0x90 0x0 ;
2: 0x90 0x0 ;
3: 0x0 0x0 0x90 0x0 ;
4: 0x6a 0x85 ;
5: 0x90 0x0 ;
6: 0x0 0x64 0x90 0x0 ;
7: 0x90 0x0 ;
8: 0x0 0x32 0x90 0x0 ;
```

```
9: 0x6a 0x83 ;
10: 0x0 0x32 0x90 0x0 ;
11: 0x6a 0x85 ;
12: 0x0 0x32 0x90 0x0 ;
13: 0x6a 0x83 ;
14: 0x0 0x32 0x90 0x0 ;
15: 0x90 0x0 ;
16: 0x63 0x1 ;
17: 0x63 0x0 ;
18: 0x90 0x0 ;
19: 0x67 0x0 ;
20: 0x0 0x32 0x90 0x0 ;
<DEBUG>No further input data (APDUs), shutting down JCOS!
```

Calculator Applet

The following shows the communication with our own test applet (see description in Section 4.4.7). The first ingoing APDU (input line 1) selects the calculator applet. The APDUs in line 2, 3, 5 and 6 are used to calculate the sum, difference, power and factorial of the two respective one (for factorial) values in the data segment. Each value occupies two bytes (short). The APDU in line 4 tests the exception handling by using a invalid instruction. This leads to an array access out of bounds and an exception is thrown. The result is an error (output line 8). The other commands generate an output containing a result (output line 6, 7, 9 and 10) stored in the first two byte of the data segment. Successful commands are always acknowledged by a status word (`0x90 0x0`).

Input (APDUs):

```
1: 0x00 0xa4 0x04 0x00 0x07 0xA0 0x0 0x0 0x0 0x62 0xFF 0x4 0x7F;
2: 0x00 0x00 0x00 0x00 0x04 0x00 0xFF 0x00 0x03 0x7F;
3: 0x00 0x01 0x00 0x00 0x04 0x00 0x02 0x00 0x04 0x7F;
4: 0x00 0x05 0x00 0x00 0x04 0x00 0x02 0x00 0x03 0x7F;
5: 0x00 0x02 0x00 0x00 0x04 0x00 0x02 0x00 0x03 0x7F;
6: 0x00 0x03 0x00 0x00 0x04 0x00 0x04 0x00 0x00 0x7F;
```

Output (APDUs + debug messages):

```
1: <DEBUG>Starting JCEP on x86!
2: <DEBUG>Starting CardOS
3: <DEBUG>VM not previously initialized, 8 packages loaded
```

```
4: <DEBUG>Start VM (card manager)
5: 0x90 0x0 ;
6: 0x1 0x2 0x90 0x0 ;
7: 0xff 0xfe 0x90 0x0 ;
8: 0x6f 0x0 ;
9: 0x0 0x8 0x90 0x0 ;
10: 0x0 0x18 0x90 0x0 ;
11: <DEBUG>No further input data (APDUs), shutting down JCOS!
```

5.2 Nonfunctional Requirements

According to Bajpai and Gorthi [35] the two design goals **Portability** and **Maintainability** (described in detail in Section 3.1) have to be treated as non-functional requirements (NFR). Due to NFRs are difficult to specify and even more difficult to measure, we haven't defined test cases for this two attributes, as we did for the functional requirements. Instead will use the next two subsection to explain why we think our implementation fulfills this two NFR.

5.2.1 Portability

Portability is one of our main design goals we stated in Section 3.1. We demanded from our implementation to be able to run on various platforms ranging from desktop workstations to micro controller. The changes need to be done when moving our implementation to a new platform should be as few as possible.

As stated before Portability is a NFR and therefor it is difficult to measure. We will use two different approaches to show the portability of our implementation. First we will discuss our experience from porting our implementation from our development platform (GCC and x86) to the 8051 platform (Keil Compiler and Simulator). Due to the fact that this is only a single case we will also show how our implementation supports future porting efforts.

8051 Port

During the implementation phase of our work we mainly used GCC (compiler and debugger). To be able to run the code on a PC we also implemented a specific HAL for this platform (directory `arch_x86`).

Mooney [36] defines the degree portability as a relation of the costs for adapting an existing implementation and the costs for developing a new one (see Equation (5.1)). One would be perfect portability thereby a positive values indicates that porting is cheaper

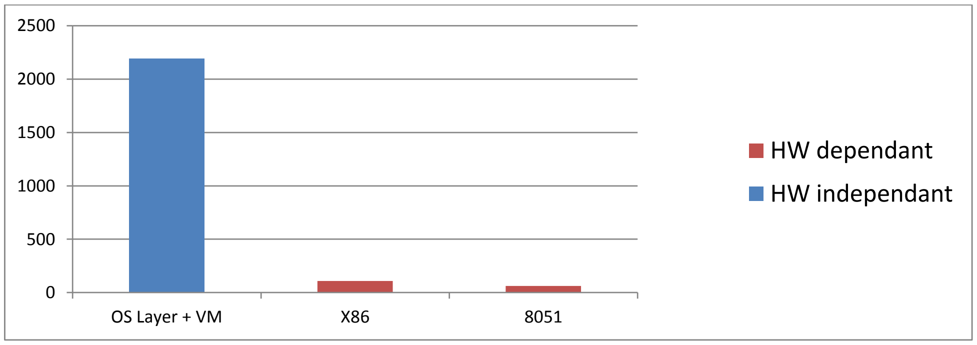


Figure 5.2: This figure show the relation between the Lines of Code (LOC) which depends on the HW (HAL for x86 and 8051) and the LOC which are HW independent (OS Layer and VM).

than redeveloping. For our implementation (using LOC as scale) the formula would lead to a degree of portability of **0.97**.

$$DP = 1 - \frac{cost\ to\ port}{cost\ to\ redevelop} \tag{5.1}$$

Figure 5.2 show the relation of platform dependent and platform independent LOC. The HW dependent part only accounts for a small part of the overall LOC. This result is reached by isolating all external interfaces in few modules (HAL and header file). Thereby porting our implementation for the 8051 platform only requires a little amount of code to be redeveloped (see also Figure 5.2).

Support for Future Porting

To aid porting to yet unknown platforms, Brown [24] recommends to use a standardized HLL and to stick to a subset of this language which is supported by most compilers. We took account for this suggestions and selected C (in the revision C90) as our programming language of choice. C is a ISO standardized language which is also popular in the smart card domain [21, Chapter 13]. The recommended subset of language features was C90 which is a matured standard supported by most compilers. Brown also recommends to use preprocessor statements to mask different platform behavior, which we implemented in `jcos_common.h`.

Mooney [36] demands to identify all external interfaces (which includes I/O and memory management) and to isolate them or use a platform overarching standard. Since no such standard (like POSIX for Unix like OS) exists for our domain, we isolated all external interfaces in the HAL and provided a well-defined interface for other SW parts (`hal_interface.h`). Mooney also mandates for using a well-standardized programming language, stating C as one example, which was also our language of choice. Another point he raised is the reusability of test cases to avoid introducing errors during porting. We achieve this goal by providing a reusable test case for HAL implementation (file `hal_test.c`) and HW independent test cases for the other parts (*googletest* framework).

5.2.2 Maintainability

Several methods were proposed to measure maintainability. One example would be the *Maintainability Index (MI)* [37] which is a fitting function for known good maintainable systems. It is based on several source code metrics and provides a single numeric value which should indicate the degree of maintainability. However the Maintainability Index was criticized by others [38] for not being very useful for developers who want to increase the maintainability of their source code. For example the index provides no hints what changes should be applied.

Instead Heitlager et. al. [38] supposes a new method for measuring maintainability based on some SW characteristics defined in ISO 9126. Subsequently they define a set of source code properties they believe to be good indicators for maintainability. In the following we will examine our implementation according to this properties. Each property earns a grade from very good (marked `++`) to very poor (marked as `--`). Unfortunately we didn't have access to all tools or metrics they used to calculate their source code properties. The often used term *unit* refers to the smallest executable part of the system. For languages like C or Java this would be a function respective a method.

Attribute	**Threshold**	**JCOS**	**Grade**
Volume	$<$ 8 man year	$<$ 1 man year	`++`
Complexity per Unit	$<$ 25% LOC with $CC > 10$	2.75% LOC with $CC > 10$	`++`
Duplication	$<$ 3% LOC	0.59% LOC	`++`
Unit Size	NA	74% LOC in units $<$ 30 LOC	NA
Unit Testing Coverage	$>$ 95% LOC	NA	NA

Table 5.1: This table shows maintainability attributes according to Heitlager et. al. [38]. The threshold is always for the best grade. For unit size they do not supply a concrete value. They also do not name the tools used to calculate the values, thus we cannot reproduce some metrics.

Volume

According to Heitlager et. al. the overall number of Lines of Code (LOC) accounts a lot to the maintainability of a SW system. Smaller systems are easier to overlook and thus also to maintain. This property isn't really relevant for us as they consider a system developed in eight man-years or less as very good. This border is much higher than the amount of time spend during a master thesis.

Complexity per Unit

The complexity of units is measured using the *Cyclomatic Complexity (CC)* [39]. The grading of the overall system depends of the proportion of LOC in complex units, and thus high risk units, in contrast to LOC of the overall systems.

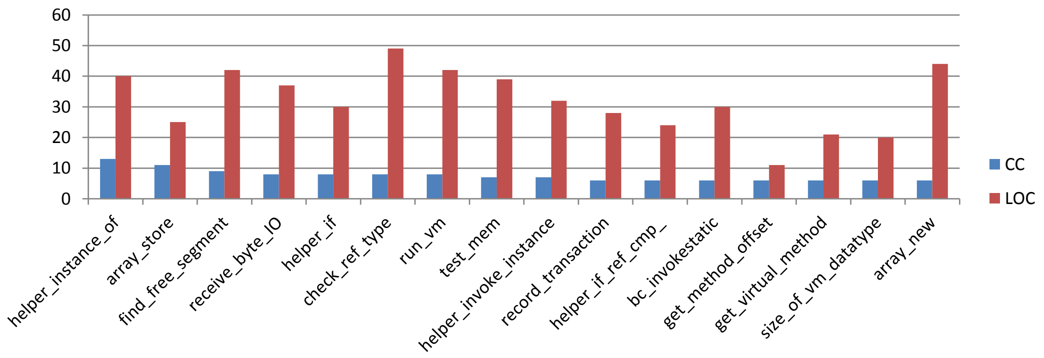


Figure 5.3: This figure shows the Cyclomatic Complexity (CC) and the Lines of Code (LOC) for some units (functions) of our implementation. Units with a CC with less than 6 are omitted

Figure 5.3 shows the CC for some units (functions) of our implementation (units with less than 6 CC are omitted). Heitlager et. al. regards units with CC of ten or lower as simple units without much risk and units with CC from 11 to 20 as units with moderate risk. If the number of line of codes in moderate risk units is below 25 percent, this property is rated very good.

Duplication

Blocks of six lines or more which occur multiple times are regarded as code duplication (according to Heitlager et. al.). Five ore less percentage of duplicated code is regarded a good value concerning maintainability.

We used *Atomiq*[1] to analyze code duplication in our source code. Looking for blocks of six lines or more, 0.59% duplicated lines were found. To validate the results of the test tool we also calculated duplication for blocks of two lines leading to 5.16% duplicated lines being found.

Unit Size

Heitlager et. al. don't give concrete thresholds for this point but states that larger units are more difficult to maintain than small one. We took account of this by trying to split large functions into smaller ones even if this results in lower execution due to more frequent function calls.

In Figure 5.4 the distribution for LOC per unit is shown. Most units only have few LOC and can be regarded as easily to maintain. Only a small part of the units are more complex (in terms of code size).

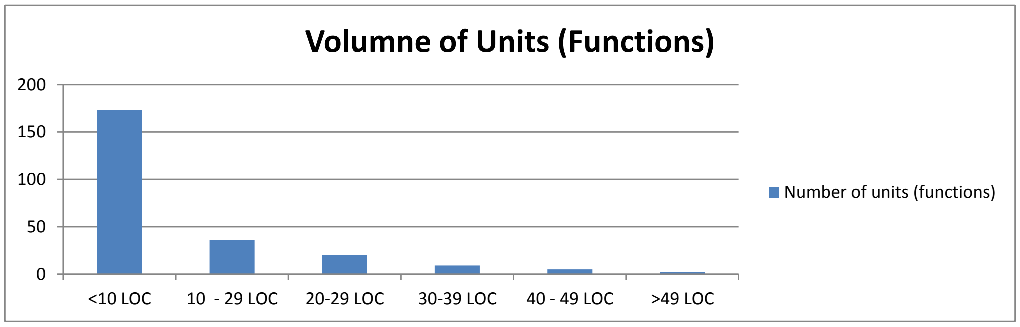


Figure 5.4: This figure shows a histogram for the unit size (function) measured in LOC.

Unit Testing

Unit testing is regarded as an important point for maintainability which we take into account by using *googletest*. See Section 3.5.2 for the description of the framework, Section 4.4 for the test cases and Section 5.1 for the results.

5.3 Summary

In this chapter we recalled the design goals we stated in Section 3.1 and determined if and how good our implementation met these goals. The first design goal, the **Compliance**

[1]Homepage `http://getatomiq.com/`

with the Java Card platform standard, was regarded as a functional requirement. In Section 4.4 we defined test cases, both unit tests and system test, to check the behavior of our implementation. The results of these teat cases are documented in Section 5.1.

The two other design goals, **Portability** and **Maintainability**, are non-functional requirements (NFR)s. NFR are difficult to cover with test cases and hard to measure. In Section 5.2.1 and Section 5.2.2 we determined the degree of portability and maintainability of our implementation by methods suggested in specialist literature. Most notable for portability were the use of a wide supported language (subset) and a small part of LOC which needed to be adopted for different platforms. For a good maintainability, an important point is the small size per unit (function) and the low complexity (measured by CC).

Chapter 6

Conclusion and Future Work

In the following chapter we will come up with our lesson learned during this master thesis. Subsequently we will conclude with an outlook to possible improvements and enhancements.

6.1 Conclusion

The conclusion will be centered around our three design goals. We will discuss the efforts we undertook to fulfill them and how close we get.

6.1.1 Compliance

The compliance to the Java Card Specification may seems to be the easiest of the three design goals as we can treat it as a functional requirement. But large functional requirements pose their specific problems; they can usually not be verified completely. In our case, the specification consists a high level description in english and a reference implementation for the PC platform. Non of them can be used for an exhaustive compliance check. A specification in a natural language always has some obscurities and ambiguities. The implementation can be used to compare specific input sets.

Because of this limitations we decided to use a mixed approach. We used the high level description to derive requirements for our unit tests and we used the reference implementation to have a *golden device* to which we can compare our own implementation.

6.1.2 Portability

Getting closer to the 'real' problems of software engineering, portability confronts us with a non-functional requirements (NFR). NFR are even worse than functional requirements as they are difficult or sometimes even impossible to measure. Portability is not the worst one but metering also poses some challenges. We used two approaches: we described our experiences during porting our implementation to an alternative platform and we discussed the measures we conducted to aid future porting efforts.

The first approach includes comparing the development efforts for different platforms (estimated using the LOC). We were able to show that only a minor part of our code is platform dependent and the rest can be reused without much adaptions. Having this number also allows us to calculate the Degree of Portability. Our implementation results in a DP of 0.97 which indicated a very good portability (one would be perfect portability, values lesser zero advise a redevelopment).

6.1.3 Maintainability

Basically measuring maintainability yields the same problems than measuring portability but we had one huge advancement; we hadn't any real experiences in maintaining our implementation because we are only at the end of the implementation phase. Due to this circumstances we decided to use the *Sig Maintainability Model*. This model defines five different attributes related to maintainability. The attributes themselves are derived from an ISO norm to increase the reliability of the values.

Although these attributes give a good overview of how good a SW system can be maintained it has some serious drawbacks (which are also addressed by the creators of the model). One example would be the measurement of unit complexity which is done by using the Cyclomatic Complexity (CC). CC grades the complexity of a unit (usually a function or a method) by the number of possible execution branches. This leads to a low complexity measure for function with a lot of unconditional calculation.

6.2 Future Work

Due to the design goals **Portability** and **Maintainability** two main fields for improvements can be identified: porting the JCOS to new platforms and improving or enhancing the source code.

6.2.1 Future Implementation Work

The possibility for functional enhancements dates back to the Java Card platform specification which only describes a minimum functionality an implementation must provide. First their are some optional features described by the specification which can be implemented in future like integer support or additional JCRE classes. Additionally implementers are free to extent their platforms with features not described in the specification (as long as they do not interfere with mandatory features). Examples for such features may be a garbage collector or additional communication protocols.

6.2.2 Future Research

Further research can explore possibilities for enhanced security on smart card platforms like defensive virtual machines, which check byte code sanity during execution or use HW features to encapsulate applet data.

Another research focus could be porting our implementation to new platforms like ARM compatible CPUs. This platform is used a lot for embedded systems and also gains popularity in the field of smart cards. Another aspect of the ARM platform is the good compiler support. ARM can be targeted by the GCC which we used intensely during our work. Thus it is likely to port the JCOS to an ARM platform with very little effort: all compiler directives can be reused, only the HAL would need adaptions.

Appendix A

Abbreviations

ABI Application Binary Interface

AID application identifier

APDU Application Protocol Data Unit

CAD Card Acceptance Device

CAP Java Card converted applet

CC Cyclomatic Complexity

CDT C/C++ Development Tools

COS Card Operating System

CPU Central Processing Unit

GCC GNU Compiler Collection

HAL Hardware Abstraction Layer

HLL-VM High-Level Language Virtual Machine

HLL High Level Language

HW Hardware

ISA Instruction Set Architecture

ISO International Organization for Standardization

JCDK Java Card Development Kit

JCOS	Java Card Operating System
JCRE	Java Card Runtime Enviroment
JCVM	Java Card Virtual Machine
JDT	Java Development Tools
LLVM	Low Level Virtual Machine
LOC	Lines of Code
MI	Maintainability Index
MMU	Memory Management Unit
NFR	non-functional requirements
OS	Operating System
PC	Personal Computer
RE	Runtime Environment
SW	Software
SoC	System on Chip
VM	Virtual Machine

Appendix B

How To

This chapter will explain how to build, run and test our implemention for users not familar with the source code.

B.1 Build

The source code was developed using Eclipse so for a first build it is the easiest way to import the source code in your Eclipse working space. It should work on all C compiler supporting the C90 standard, we tested it on MinGW and GCC.

To build the code using a Eclipse, an empty C project has to be created. Subsequently the two directories `scr` and `include` have to be added to the project. The `include` directory also has to be in the search path for header files (for Eclipse: Project Settings → C/C++ General → Paths and Symbols).

During development we recommend to built the code using the DEBUG flag (see Table B.1 for all preprocessor options). When set, parts of the C standard library are included and several debug messages are printed on stdout. This may not work an all platforms/compilers or interferes with the APDU channel (depending on the HAL).

Flag	**Value**	**Description**
DEBUG	NA	print debug messages, link with stdlib
ARCH	X86, 8051 ...	define platform, if not defined X86 is used
OMIT_MAIN	NA	remove main function in HAL, needed for test framwork
TEST_HAL	NA.	only run HAL test (see subsection 4.4.1)

Table B.1: This table shows all available global preprocessor options.

B.2 Run

How to run the JCVM strongly depends on the used platform and how the specific HAL is implemented. When using the x86 HAL the JCVM can be started like a normal executable. Two parameters have to be supplied: a binary file which can be used to initialize the persistent memory and a text file which contains incoming APDUs.

```
<DEBUG>Usage: jcep <pmem-image> <apdu-file>
```

The incoming APDUs have to be encoded in hex values, separated by blanks. One line represents one APDU and has to be terminated with a semicolon. The same format was used in subsection 5.1.7 to show the results of the test applets.

The outgoing APDUs are stored in the file `apdu.out`. This file uses the same format as previously described. During shutdown the state of the persistent memory is stored in the file `pmem.new`.

Installing Additional Applets

JCOS currently doesn't contains an on-card installer as this feature is not requested by the Java Card specification. Nethertheless new applets can added in the same way we included the JCRE packages and the test applets. The function `load_jcre_packages` in file `packages.c` loads and initializes this packages. First a CAP file struct has to be created and the pointer to the different components have to be set. Subsequently it can be added to be list of available packages and the static fields can be initialized using the function `init_static_field`. If the package contains an applet, `install_applet` can be called to install it. At the end, the number of installed packages has to be updated (`cap_container_s.count`).

B.3 Test

Generally speaking, three different kinds test cases exist for (parts of) the JCVM. The HAL is tested by a special module, most other parts are tested using googletest and the whole systems can be tested by a number of sample applets.

B.3.1 HAL Test

To test a specific HAL, the JCVM has to compiled with the flag `TEST_HAL` (see Table B.1). When the executable is deployed on the target platform it launches a test routine (in source file `hal_test.c`) for the HAL instead of the VM.

For the x86 HAL just start the executable and see how memory access is checked and the input is echoed.

B.3.2 Unit Test

To run the unit tests simple compile the containing Eclipse project (`jcep_testing`) and run the executable. The projects have to be in the same workspace and the platform specific googletest libraries have to copied into the `lib` directory. The unit test can only be performed on platforms supported by a C++ compiler and an available stdout stream (like x86/Linux).

Appendix C

Naming Conventions

Throughout the the source file we used the following rules to name variables, functions and methods. For historical reasons we use different naming conventions for Java and C source code.

C.1 C

In C source and header files names usually contain only lower case letters and are separated by underlines (e.g. `my_variable`). Names are chosen using the following rules:

- **Structs, Unions and respective pointers**
  Structs are like normal names but alway contain `_s` at the end, respective unions end with `_u`. A typedef pointer to any type always ends with `_p`.
- **Local and global Variables**
  Variables are named obeying the general rule, only lower case letters separated by underlines. Global variables have an additional underscore at the end (e.g. `my_global_var_`).
- **Function Names**
  No special rules apply for function names, we only try to chose descriptive names.
- **Preprocessor Statements**
  Preprocessor statements are an exception to the general use of lower case letter and have names using only upper case letters and underscores. This stamens include constants and macros, both created using `#define` (e.g. `VM_STACK_SIZE`).

C.2 Java

As mentioned before for historical reasons we choose a different naming convention for Java code in our implementation. We will stick largely to the coding standard for java provided by Oracle [40]. It request class names to begin with a capital letter and variable and method names with lower case letters. Name are separated by using capital letters inside an name (e.g. `MyClass` or `methodForMyClass`).

Appendix D

Applet Scripts

D.1 Wallet

The input script below is used to access the *Wallet* supplied by the JCDK (text is taken from file `wallet.scr`)..

Input (APDUs + comment lines with trailing //):

```
/////////////////////////////////////////////////////////////////////
// Initialize Wallet
/////////////////////////////////////////////////////////////////////

//Select Wallet
1: 0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6 0x1 0x7F;
// 90 00 = SW_NO_ERROR

//Verify user pin
2: 0x80 0x20 0x00 0x00 0x05 0x01 0x02 0x03 0x04 0x05 0x7F;
//90 00 = SW_NO_ERROR

//Get wallet balance
3: 0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x00 0x00 0x00 0x90 0x00 = Balance = 0 and SW_ON_ERROR

//Attempt to debit from an empty account
4: 0x80 0x40 0x00 0x00 0x01 0x64 0x7F;
//0x6A85 = SW_NEGATIVE_BALANCE
```

```
//Credit $100 to the empty account
5: 0x80 0x30 0x00 0x00 0x01 0x64 0x7F;
//0x9000 = SW_NO_ERROR

//Get Balance
6: 0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x64 0x9000 = Balance = 100 and SW_NO_ERROR

//Debit $50 from the account
7: 0x80 0x40 0x00 0x00 0x01 0x32 0x7F;
//0x9000 = SW_NO_ERROR

//Get Balance
8: 0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

//Credit $128 to the account
9: 0x80 0x30 0x00 0x00 0x01 0x80 0x7F;
//0x6A83 = SW_INVALID_TRANSACTION_AMOUNT

//Get Balance
10: 0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

//Debit $51 from the account
11: 0x80 0x40 0x00 0x00 0x01 0x33 0x7F;
//0x6A85 = SW_NEGATIVE_BALANCE

//Get Balance
12: 0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

//Debit $128 from the account
13: 0x80 0x40 0x00 0x00 0x01 0x80 0x7F;
//0x6A83 = SW_INVALID_TRANSACTION_AMOUNT

//Get Balance
14: 0x80 0x50 0x00 0x00 0x00 0x02;
```

```
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

//Reselect Wallet applet so that userpin is reset
15: 0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6 0x1 0x7F;
// 90 00 = SW_NO_ERROR

//Credit $127 to the account before pin verification
16: 0x80 0x30 0x00 0x00 0x01 0x7F 0x7F;
//0x6301 = SW_PIN_VERIFICATION_REQUIRED

//Verify User pin with wrong pin value
17: 0x80 0x20 0x00 0x00 0x04 0x01 0x03 0x02 0x66 0x7F;
//0x6300 = SW_VERIFICATION_FAILED

//Verify user pin again with correct pin value
//0x80 0x20 0x00 0x00 0x08 0xF2 0x34 0x12 0x34 0x56 0x10 0x01 0x01 0x7F;
18: 0x80 0x20 0x00 0x00 0x05 0x01 0x02 0x03 0x04 0x05 0x7F;
//0x9000 = SW_NO_ERROR

//Get balance with incorrect LE value
19: 0x80 0x50 0x00 0x00 0x00 0x01;
//0x6700 = ISO7816.SW_WRONG_LENGTH

//Get balance
20: 0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR
```

Bibliography

[1] CoCoon, "CoCoon Project Webpage." online, 2013.

[2] S. Yoo and A. A. Jerraya, "Introduction to hardware abstraction layers for SoC," in *Embedded Software for SoC*, pp. 179–186, Springer, 2004.

[3] W. Ecker, W. Müller, and R. Dömer, *Hardware-Dependent Software: Principles and Practice.* Springer London, Limited, 2009.

[4] A. S. Tanenbaum, *Modern Operating Systems.* Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2007.

[5] A. Kadav and M. M. Swift, "Understanding Modern Device Drivers," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pp. 87–98, ACM, 2012.

[6] V. Handziski, J. Polastre, J. Hauer, C. Sharp, A. Wolisz, and D. Culler, "Flexible hardware abstraction for wireless sensor networks," in *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pp. 145–157, 2005.

[7] A. Aho, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley series in computer science, Pearson/Addison Wesley, 2007.

[8] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[9] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann, 2005.

[10] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification: Java Se, 7 Ed.* Always learning, Prentice Hall PTR, 2013.

[11] D. Box and C. Sells, *Essential.NET: Volume 1: the Common Language Runtime.* Essential .NET, ADDISON WESLEY Publishing Company Incorporated, 2003.

[12] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, IEEE, 2004.

[13] A. Zakai, "Emscripten: An LLVM-to-JavaScript Compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 301–312, ACM, 2011.

[14] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 79–93, IEEE, 2009.

[15] A. Donovan, R. Muth, B. Chen, and D. Sehr, "PNaCl: Portable Native Client Executables," 2011.

[16] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov, "Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices," in *Sensor Technologies and Applications, 2009. SENSORCOMM'09. Third International Conference on*, pp. 117–125, IEEE, 2009.

[17] N. Shaylor, D. N. Simon, and W. R. Bush, "A Java Virtual Machine Architecture for Very Small Devices," in *ACM SIGPLAN Notices*, vol. 38, pp. 34–41, ACM, 2003.

[18] I. L. Marques, J. Ronan, and N. S. Rosa, "TinyReef: a register-based virtual machine for Wireless Sensor Networks," in *Sensors, 2009 IEEE*, pp. 1423–1426, IEEE, 2009.

[19] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 85–95, Oct. 2002.

[20] P. Stanley-Marbell and L. Iftode, "Scylla: A smart virtual machine for mobile embedded systems," in *Mobile Computing Systems and Applications, 2000 Third IEEE Workshop on.*, pp. 41–50, IEEE, 2000.

[21] W. Rankl and W. Effing, *Smart Card Handbook.* Wiley, 2010.

[22] Oracle, *Virtual Machine Specification.* Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[23] Oracle, *Runtime Environment Specification.* Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[24] P. J. Brown, "Software Portability," in *Encyclopedia of Computer Science*, pp. 1633–1634, Chichester, UK: John Wiley and Sons Ltd., 2003.

[25] Oracle, "Java Card Classes."

[26] R. D. M. Kernighan Brian W., *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd ed., 1988.

[27] "Eclipse Wiki." http://wiki.eclipse.org/.

[28] "About the Eclipse Foundation." http://www.eclipse.org/org/.

[29] "GNU Compiler Collection Documentation." http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc.pdf.

[30] "Netbeans." https://netbeans.org/.

[31] "Java Card Development Kit Documentation."

[32] "codavaj at sourceforge.com." http://codavaj.sourceforge.net/.

[33] "Keil Getting Started Guide." http://www.keil.com/product/brochures/uv4.pdf.

[34] "googletest Wiki." https://code.google.com/p/googletest/wiki.

[35] V. Bajpai and R. Gorthi, "On non-functional requirements: A survey," in *Electrical, Electronics and Computer Science (SCEECS), 2012 IEEE Students' Conference on*, pp. 1–4, 2012.

[36] J. D. Mooney, "Bringing portability to the software process," 1997.

[37] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[38] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pp. 30–39, 2007.

[39] T. J. McCabe, "A Complexity Measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[40] Oracle, "Java Coding Standard." http://www.oracle.com/technetwork/java/codeconv-138413.html.