Master Thesis

Application-based Evaluation of MPSoC-Smartcard Communication Architectures

Matthias Michael Viertler

Institute for Technical Informatics
Graz University of Technology
Chairman: Univ.-Prof. Dipl.-Inform. Dr. Kay Römer

In Cooperation with
NXP Semiconductors Austria GmbH

Assessor: Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
Advisor: Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
Dipl.-Ing. Dr. techn. Christoph Trummer

Graz, October 2014

Kurzfassung

Aufgrund der Entwicklung heutiger System-on-Chips (SoCs), immer mehr Komplexität auf einem einzelnen Chip unterzubringen, steigt auch die Anzahl der integrierten Hardware (HW) Komponenten. Da diese Komponenten zumeist auch untereinander kommunizieren, entsteht zusätzlicher Kommunikationsaufwand. Überraschenderweise ist es bei manchen Applikationen deshalb nicht die Central Processing Unit (CPU), welche den Flaschenhals des Gesamtsystems darstellt, sondern die Verbindungsarchitektur, welche für die Kommunikation der einzelnen Module miteinander verantwortlich ist.
Eine ähnliche Entwicklung konnte im Bereich der eingebetteten Systeme und einem Teilbereich dessen - den so genannten Smartcards - beobachtet werden. Heutzutage kommt es zu einer ansteigenden Miniaturisierung der HW Komponenten, wodurch komplett autonome Sub-Systeme, jedes für eine spezielle Aufgabe bestimmt und auf einem eigenen Chip untergebracht, auf einer Multi-Processor System-on-Chip (MPSoC)-Smartcard integriert werden können. Der Fokus dieser Arbeit liegt auf der Kommunikationsinfrastruktur zwischen diesenTeilsystemen, da die Kommunikation innerhalb der Teilsysteme zumeist Bus-basiert ist.
Smartcards werden oft für einen ganz bestimmten Zweck oder eine ganz bestimmte Aufgabe entworfen. Natürlich führen diese unterschiedlichen Anwendungen zu unterschiedlichen Kommunikationsprofilen. Die Datenmenge und Kommunikationsfrequenz variiert stark, abhängig von der betrachteten Applikation, was eine allgemeingültige Kommunikationsarchitektur für alle Applikationen ausschließt.
Typische Smartcard Applikationen sind unter anderem Bank- und Zahlungsanwendungen, so genanntes Smart Metering oder Software (SW) Update eines Applets, jede mit unterschiedlichem Kommunikationsverhalten und zeitlichen Anforderungen. Um die effizienteste Kommunikationsarchitektur für eine bestimmte Anwendung auswählen zu können, müssen jene evaluiert werden, welche den gebräuchlichsten Szenarien entsprechen. Dies wird normalerweise mittels Simulationen zur Evaluierung des Datendurchsatzes bewerkstelligt, also dem Modellieren von verschiedenen Kommunikationsarchitekturen und anschließendem Ausführen einer abstrakten Version einer Anwendung auf diesen Architekturen. Da Smartcards jedoch zusätzlichen Einschränkungen unterliegen, ist eine Klassifikation der Architekturen aufgrund des Durchsatzes alleine unzureichend.
Deshalb müssen das Design und die Implementierung der Kommunikationsarchitekturen und Smartcard-Anwendungen alle relevanten Einschränkungen erfassen und korrekt abbilden, um diejenige Architektur zu finden, welche für eine bestimmte Anwendung am angemessensten ist. Die Resultate werden den Restriktionen einer Anwendung entsprechend interpretiert, um am Ende der Arbeit eine Schlussfolgerung über die passende Kommunikationsarchitektur für eine bestimmte Anwendung erbringen zu können.

Abstract

The evolution of today's System-on-Chips (SoCs) towards ever greater complexity on a single chip implies that the number of integrated Hardware (HW) components is rising. As these components often interact with each other, additional on-chip communication has been introduced. Surprisingly, for some applications it is not the Central Processing Unit (CPU) being the performance bottleneck anymore, but the communication infrastructure that is responsible for connecting the modules to each other.
A similar development is taking place in the field of embedded systems and its subgroup: smartcards. Nowadays, the increasing miniaturization of the HW components allows completely autonomous sub-systems - each of them performing some dedicated work and being placed on its own chip - to be integrated in an Multi-Processor System-on-Chip (MPSoC)-Smartcard. The focus of this work is the communication infrastructure between these sub-systems as the communication inside the sub-systems is considered to be bus-based.
Smartcards are often designed for a specific purpose or application. Naturally, these differing applications also lead to diverse communication demands between the single chip-components. Data amount and frequency of the applications' communication scenarios describe a significant range, thus a generic communication architecture solution is not suitable for all applications.
Among others, typical smartcard applications are banking, smart metering or a Software (SW) update of an applet, each of them having different communication behaviors and timing-limits. In order to be able to choose the most-efficient communication architecture for a specific application's communication profile, evaluations have to be performed, covering the most common scenarios. This is typically done with simulation-based performance evaluations, modelling the various communication architectures and executing an abstract version of specific applications on them. However, as smartcards are subject to various additional constraints, these evaluations cannot consider the performance aspect alone. For a proper choice of which architecture to use for which application, at least an architecture's area-needs and power-dissipation have to be considered as well.
Therefore, the design and implementation of the communication architectures and smartcard applications have to capture all relevant constraints to evaluate the best-fitting architecture for a specific smartcard application. Depending on each application's communication profile and timing-constraints, the simulation-results are interpreted and a proper conclusion about which architecture to use for which application is given at the end of this thesis.

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)

Acknowledgments

This master thesis has been written in 2014 on the *Institute for Technical Informatics (ITI)* at the Graz, University of Technology.

I would like to thank all from the ITI - especially Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger who was the assessor as well as advisor for this work at the university. Additionally, I would like to thank the *NXP Semiconductors Austria GmbH* company and here especially Dipl.-Ing. Dr. techn. Christoph Trummer who was this work's advisor.

Further I would like to thank my parents - Maria and Friedrich - for their support throughout the years.

Graz, October 2014 Matthias Michael Viertler

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The tasks of modern embedded systems are becoming more and more complex. Therefore, the evolution of embedded systems is towards so-called Multi-Processor System-on-Chips (MPSoCs). In order to handle the increasing complexity of the implemented applications, the number of processors integrated into such a system is rising.

Embedded systems often adopt the evolution of high-end solutions with some delay. Therefore, modern embedded systems also consist of multiple processors. Each of them located in a sub-system dedicated for a specific sub-task such as representing the communication interface to external systems, performing measurements, etc. Additionally, these sub-systems also often have co-processors attached for a further increase in performance. These co-processors are responsible for e.g. cryptographic-, Cyclic Redundancy Check (CRC)- or Random Number Generator (RNG)-calculations.

It is obvious that all these systems and sub-systems have a very high communication demand among each other which is becoming the bottleneck of modern embedded systems. As the communication inside these sub-systems among the microprocessor, its memory and co-processor is usually bus-based, it is the communication infrastructure between the sub-systems that is of interest. Therefore, different communication architecture approaches have to be evaluated in order to find the most efficient for a specific application.

In embedded systems, these communication architectures not only have to be appraised on the aspect of performance alone but also with respect to their HW-costs (area) and power-efficiency. Especially, when it comes to passively-powered embedded systems such as smart cards. These systems have very limited power-ressources and thus, cannot afford to spend too much of their energy on the communication alone.
Since smartcards - and also embedded systems in general - are often tailored to one specific task or application, the most efficient communication architecture will be highly dependent on the communication scenario of the concrete task or application.

The focused smartcard applications will cover different traffic-scenarios and thus, allow a valuable conclusion on which communication approach one should prefer for which system. Explicit applications of interest are banking (see 3.2.1), smart metering (see 3.2.2) as well as secure software updates (see 3.2.3) of e.g. the smartcard's Operating System (OS).

These applications cover the profiles where little data is sent often (banking), moderate data is sent quite frequently (smart metering) and very much data is sent rarely (secure-software update).

Before a detailed description of this work's objectives (see 1.3) and its underlying motivation (see 1.2) are shown, some background knowledge is presented (see 1.1), covering the fields of embedded systems as well as smartcards. Also an overview about multiprocessing is given to be able to put the target platform into the right context.

1.1 Background

1.1.1 Embedded Systems

The world is full of so-called embedded systems (see Embedded Systems) which perform very specialized tasks they have been designed for. These tasks are called embedded systems applications or embedded applications (see Embedded Applications). Before these applications can be specified further, a clear definition of an embedded system is needed.

An embedded system is a system designed to interact with its environment - either by measuring some physical dimensions, controlling specific procedures in a control systems' way or helping to automate very trivial processes with a simple underlying logic. In general, the task the system is designed for is known a priori, thus there is no need to re-program it during its life-cycle. Due to the evolution of the complexity of embedded systems' applications, more and more of the modern systems are microprocessor-based allowing to model complex routines at lower costs than with a dedicated Hardware (HW) solution. [Hea02]

Although, an embedded system can also consist of a simple HW-logic without a microprocessor, the rising complexity of today's embedded systems leads to a shifting of the proportion towards microprocessor-based systems.
The majority of the microprocessors built these days are not integrated into a desktop Personal Computer (PC) or notebook as many people would assume but in embedded systems.

Typically, embedded systems often face some constraints [Hea02]:

- Real-Time:
  A system's success depends on having its results ready after some specific time to meet some pre-defined deadlines.
- Power:
  Energy is often limited in such systems, thus the power consumption has to be considered.
- Costs:
  As the number of produced entities of an embedded system is mostly very high, the system has to be cost-effective.

- Area:
  Also the required area has to be kept in mind in order to allow the systems to be highly integrable.

Embedded Applications

Described in a very abstract way, embedded applications can be seen as systems with a number of inputs and a number of outputs, enabling them to interact with their environment. Typically, some data is measured via its inputs, then this data is being evaluated and finally some specific actions depending on these values are performed or simply a result is dumped via the outputs. [Hea02]

After this abstract definition some concrete examples are given for typical embedded applications. Common embedded application areas are as follows [Mar03]:

- Automotive electronics:
  Air-bag or engine control systems, Anti-lock Braking System (ABS), Electronic Stability Control (ESC), etc.

- Avionics:
  Flight control, anti-collision or pilot information systems, etc.

- Railways:
  Similar to car and airplane features (safety, information).

- Telecommunication:
  Mobile phones (Smartphones) with their Radio Frequency (RF), digital signal processing and low power design.

- Security:
  Secure identification / authentication of people, e.g. for passport / company access control etc.

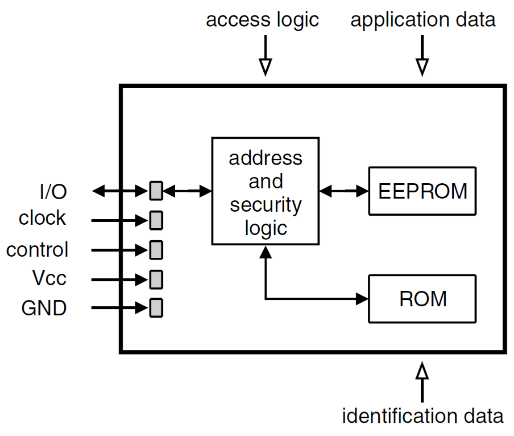
- Banking:
  Payment applications on credit / debit cards as well as via Near Field Communication (NFC) for electronic payment, etc.

- Smart Metering:
  Remote access to heating-, power- and water-meters. Due to data protection often combined with security applications.

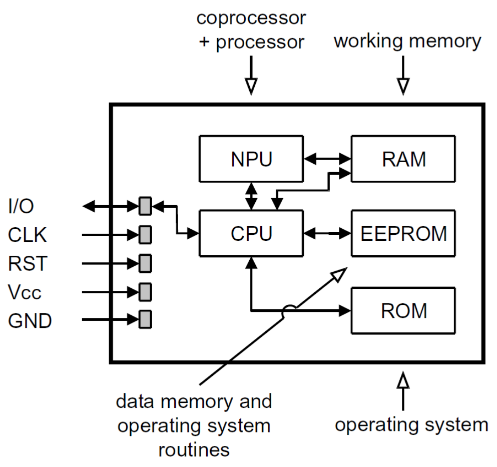
- Other Areas:
  Health sector, Fabrication Equipment, Consumer Electronics, etc.

1.1.2 Smartcards

A subgroup of embedded systems are smartcards, basically plastic cards containing some integrated logic. They are the successor of magnetic- stripe cards, which have the disadvantage that the data stored on this stripe can be manipulated by anyone with the proper



(a) Architectural overview of a memory smartcard with security logic for memory access [WR03, page 19]



(b) Architectural overview of a microprocessor card [WR03, page 20]

Figure 1.1: Smartcard Variants: Memory Card and Microprocessor Card

equipment. Although, magnetic-stripe cards still provide the basis for the majority of today's financial-transaction-related cards due to historical reasons. It is the high costs due to the online checks needed to be performed in background to provide a basic level of security that will lead to the extinction of the magnetic-stripe card.With the ongoing miniaturization of Integrated Circuits (ICs) in the 1970s it was possible to integrate both, a simple HW-logic as well as data memory on a single chip. [WR03]

Smartcard Applications

The increasing complexity that could be integrated in an IC led to identification cards being built out of such systems allowing a manipulation- resistant design. These cards are called to be *memory cards* and were used as telephone cards in the 1980s. [WR03]
The next step was to integrate a microprocessor into smartcards to further improve their functionality - the *microprocessor cards*. It was in the end of the 1980s when a predecessor of today's Universal-Integrated-Circuit-Card (UICC) - also known as *Simcard* - was used for authorization in the formerly analog mobile telephone network. [WR03]
These were the first applications of smartcards with many others following. The higher complexity of banking applications led to the delay of microprocessor-based smartcards, as the cryptographic procedures to secure these systems had to be developed first. [WR03]
Besides these two fields of applications, the area of applications covered by smartcards is enormous: medical insurance cards, digital signatures, personal identity cards, electronic toll systems, etc. [WR03]

Smartcard Architectures

Figure 1.1a shows a typical memory card consisting of an address- and security-logic where the access to the memory is controlled with. The memory consists of a Read Only Memory (ROM), holding the identification data that is written during manufacturing and cannot be overwritten or erased, as well as an Electrically Erasable Programmable Read-Only

Memory (EEPROM), holding the application data. The latter allows a rewriting of data as the name already suggests and is non-volatile, thus holds its data also when the power is turned off. On the left hand side one can see the typical pins for Input / Output (I/O) as well as power supply. [WR03]

Figure 1.1b depicts a microprocessor smartcard with a microprocessor (here: Central Processing Unit (CPU)), a co-processor (here: Numeric Processing Unit (NPU)) often used for cryptographic calculations, a ROM containing the smartcard's OS, the EEPROM used as non-volatile memory as well as the Random Access Memory (RAM) representing the working-memory used by the CPU and the NPU. [WR03]

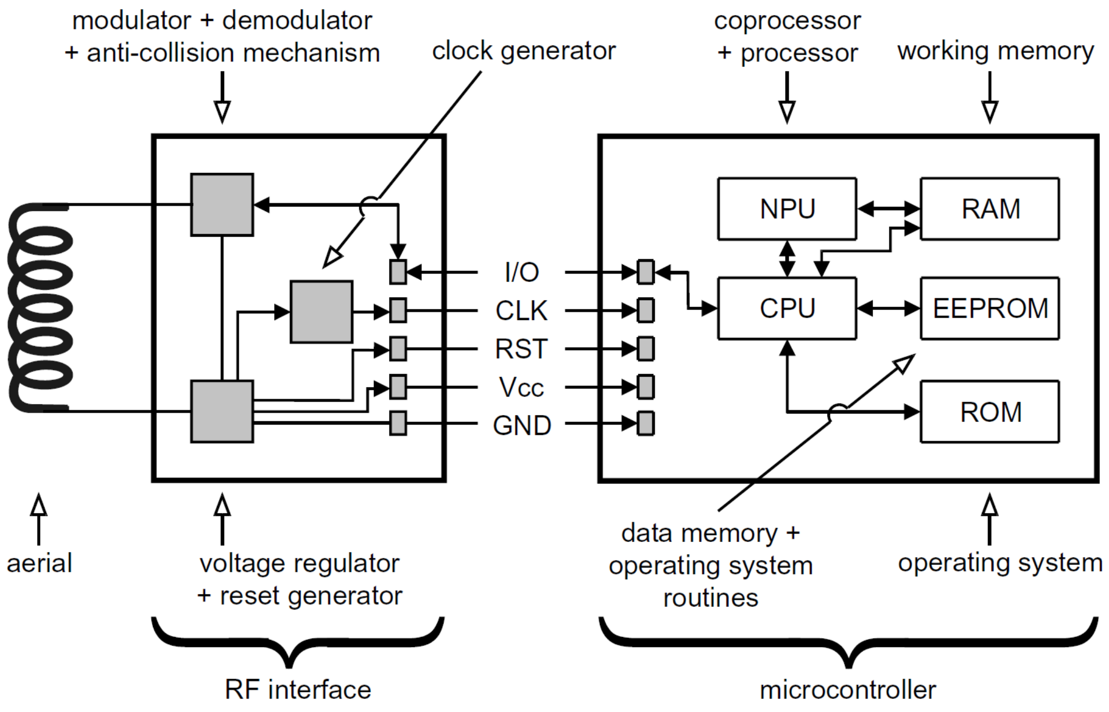


Figure 1.2: A microprocessor smartcard with a contactless interface used for communication and power supply [WR03, page 22]

An alternative data transmission method beside the contact-based one shown in the Figures 1.1a and 1.1b is the contactless interface outlined in Figure 1.2. Contactless smartcards have numerous advantages as there is no mechanical contact wear compared to contact-based smartcards. Also electrostatic discharge and the customer acceptance is higher as no particular card-insert-direction has to be considered any more.
The coil in the very left hand side of Figure 1.2 is used to receive data and energy from the induced signal sent by the transmitter device called reader, as well as to respond data through modulating the signal in terms of changing its amplitude. The anti-collision mechanism is needed if multiple smartcards are nearby, interfering with each other, allowing the reader to select one specific card for communication. A clock generator extracts a clock signal out of the incoming signal which is then used to clock the smartcard's IC. [WR03]

Smartcards can thus be summerized as systems, containing an IC with data processing, storing and transmitting functionality. Additional to their higher data storage capacity compared to magnetic-stripe cards it is most of all their ability of storing data in a secure and reliable manner that has led to their widespread usage. [WR03]

1.1.3 Flynn's Taxonomy

Flynn's Taxonomy describes the classification of parallel systems [Fly72]. According to Flynn, the computer system can be seen as a system with two streams of information, an instruction- and a data-stream. Depending on the way the underlying system manipulates these streams it can be either a Single Instruction Single Data (SISD), a Single Instruction Multiple Data (SIMD), a Multiple Instruction Single Data (MISD) or a Multiple Instruction Multiple Data (MIMD) system as depicted in Figure 1.3:

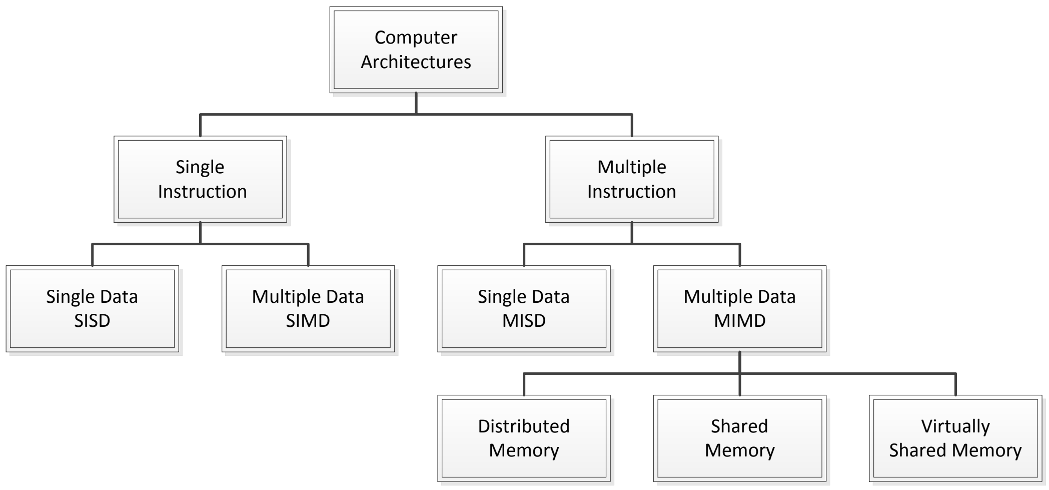


Figure 1.3: Classification of parallel systems according to Flynn [mim, page 43] and [intb]

SISD:
: The simplest system is the SISD. It is the ordinary single-processor system sequentially executing one instruction after the other. There is no parallelism as work is performed on a single instruction-, as well as single data-stream.

SIMD:
: The next system is the SIMD. Especially for parallelizable manipulations such as video-stream manipulations (rendering) where exactly the same operation is performed on multiple data (the pixels of an image).

MISD:
: The MISD system describes a system performing multiple instructions on a single

data stream which is practically relevant only for error- detection systems, where multiple processing elements do the same calculation and compare their result.

MIMD:
The most powerful system is the MIMD system. It describes a system consisting of multiple, completely autonomous, units performing different instructions on different data. These autonomous systems (each system is executing its own instructions on its own data) are subdivided into the way their memory space is organized: either *distributed memory* or *shared memory.*

For the considered embedded system target platform the focus will be on the MIMD system which can be further partitioned into the way the memory is organized [Kub97]:

Distributed Memory:
The memory is logically (and maybe also physically) distributed among the processors, thus having a system with no global address space. In such systems, message-passing communication schemes are dominant. The synchronization is then achieved implicitly through the exchanged messages.

Shared Memory:
The memory is logically (and maybe also physically) shared among the processors, thus having a system with a global address space. In such systems, memory-coupled communication schemes are dominant. The synchronization is then achieved explicitly via global-variables (e.g. mutex, semaphore, etc.).

1.1.4 Target Platform

As already mentioned in Subsection 1.1.2, the tasks handled by modern smartcards are becoming more and more complex. In order to be able to keep latencies regarded to computation, communication as well as I/O low, distinct sub-systems dedicated for this sub-tasks are needed.

Therefore, the target platform of this work's evaluation consists of two instances of the sub-system (see Figure 1.4) interconnected by some interconnection-fabric (see Figure 1.5). According to the classification scheme introduced in Subsection 1.1.3 it is this an *MIMD* system.
The sub-system is a simple Micro-Controller (μC), thus consisting of a microprocessor (Advanced RISC Machines (ARM)-CPU), a ROM holding the instructions the microprocessor executes, and a RAM used to store the data that is manipulated by the instructions being executed on the microprocessor. For reasons of clarity and comprehensibility the rest of the sub-systems' components are not shown (e.g. Watch Dog Timer (WDT), RNG, co-processor etc.).

Nowadays, on smartcards the whole communication and calculation logic is often put into a single microprocessor. Due to the rising complexity of their applications, the better scalability and the bigger potentials of cost savings, it is very likely that the system will be split up into multiple sub-systems, each of them dedicated for a specific task.

Such a sub-system is responsible for e.g. receiving commands over an external interface such as NFC, evaluating and forwarding these commands to a dedicated sub-system. Another sub-system is probably responsible for a specific task such as collecting some measured data from a connected measurement device, encrypt these data and send it back to the communication system.

Depending on the specific application and power-requirements of the embedded system, the two sub-systems could have the same but also different computation power, ROM and RAM sizes. Since the latter two factors do not affect the intercommunication architecture directly they are not further considered. However, a difference in computation power has to be considered by some sort of buffering mechanism included in the interconnection architecture to avoid - or at least limit - a blocking of either μC.

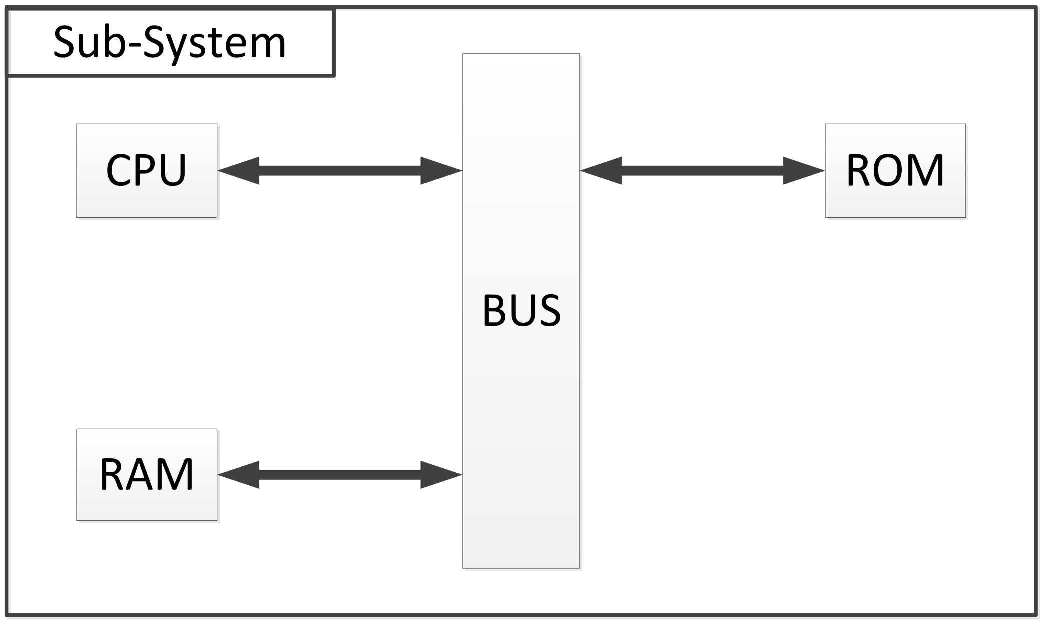


Figure 1.4: A sub system is basically a μC and responsible for dedicated sub-tasks

1.2 Motivation

As already stated in Subsection 1.1.1, various applications exist for embedded systems and their complexity is on the rise. This means the systems consist of more and more components which often have to communicate with each other. Therefore, modern embedded systems' bottlenecks are not the sole processing power of the microprocessor alone any more, but the communication infrastructure which is used by all components and often shared in a timely manner.

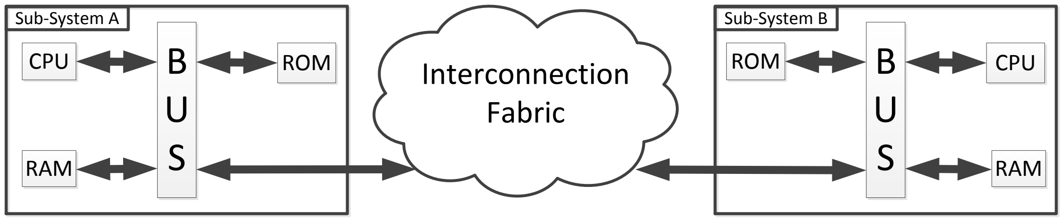


Figure 1.5: The target system consisting of two basic systems connected by some interconnection fabric

Depending on the embedded application often timing constraints have to be met, thus latencies introduced by the communication have to stay within specific limits. To fulfill the specific applications' requirements, the bandwidth limitations of the communication interconnect-fabric has to be considered.
Besides the plain performance indicator of an intercommunication architecture also the hardware costs in terms of area needed on the chip die has to be kept in mind. Interferences can be drawn from the area about the power consumption, often another important factor for embedded systems as seen in Subsection 1.1.1.

Since already depicted in Subsection 1.1.4 the targeted platform consists of two μCs which should be connected to each other. This means, the solutions' scalability and its expandability is not of much interest as the number of units to be connected is fixed. The goal is to find the appropriate communication architecture for the target applications.
However, before an evaluation of the various communication architectures can be performed, the correct abstraction layer has to be found in order to get meaningful results. Additionally, an appropriate modeling-language has to be found for this early design stage. Summarized, the complete simulation-system representing the testbench for the different approaches has to be specified.
Both communication scheme bases depicted in Subsection 1.1.3 have to be investigated in order to find the best basis for the custom HW solution. Thus, the advantages as well as the disadvantages of dedicated- and those of shared-memory approaches have to be evaluated.
Additionally, a complete benchmark system has to be specified. The question whether static or dynamic performance estimation is most applicable has to be answered. Also the language that fits best for these purposes has to be determined. Therefore, it is inevitable to find the appropriate steps needed to build a benchmark which is modelling all aspects related to the underlying application - most important with respect to its communication behavior.

In general, the question we are facing here is which communication model to use for which application? Obviously, an application where a few bytes are exchanged between the components only once in a while will not justify a HW communication module being designed for very high bandwidth data transmissions. Such a system with lots of wires for communication consumes quite a lot energy which is a limiting factor in some embedded systems like smartcards. Therefore it is extremely important to choose the right communication architecture for a specific application.

1.3 Objectives

In order to be able to answer the main question of Section 1.2, which communication architecture to use for which application and where to draw the lines, it is inevitable to perform a quantitative evaluation on the various communication architecture approaches. First of all, the right HW abstraction layer for these evaluations has to be determined. Next, a well-defined designflow to gain comparable models has to be found. Afterwards, a simulation environment has to be set up where these models can be simulated and tested against their requirements.
Representing the underlying embedded applications in benchmarks is the way to achieve convincing timing values. To cover a wide range of applications with the designed system, these application models have to be selected carefully. They should embrace as many possible use-cases as possible, ranging from little data sent often over medium data sent quite regularly to much data sent frequently. Again, to be able to do the finetuning afterwards these benchmarks have to be adaptable.
The selected applications are some of the typical applications of today's smartcards. These are: a banking application where one sub-system is the communication interface to an external terminal and one sub-system is the entity holding the secret key, a smart metering application where the measurement is performed by the second sub-system leading to some communication need when data has to be collected by an external request, and a secure software update application where some Software (SW) is updated and saved in a secure region being represented by the second sub-system.
Since the goal of this evaluation is to highlight the introduced communication overhead by a specific architecture, the benchmarks have to be on the right abstraction layer. They should cover all the properties which influence the timing behavior such as the number of transactions and data bytes per transaction as well as the delays during the communication being performed between these sub-systems.

1.4 Outline

This work is structured as follows: In Chapter 2 the related work will be discussed including an overview about communication architecture exploration (2.1) explaining a typical setup for communication architecture analysis, the right layer of abstraction and how performance analysis is done on such systems. Also some of the most popular distributed-memory based models (2.2) with a point-to-point connection fabric as well as common shared-memory based models with a bus-interconnect (2.3) are shown.
Then, in Chapter 3, the design considerations (3.1) covering the application's requirements and the simulation, benchmark as well as modules setup, are listed. Afterwards, the design of the applications that serve as templates for typical smartcard applications - covering a wide range of the communication behaviors - is presented (3.2). Also the design of the selected communication architecture approaches is shown (3.3).

The succeeding Chapter 4 is all about the implementation details for the designs selected in Chapter 3. Besides the details for the applications (4.2) and architectures (4.3), first of all the implementation considerations (4.1) regarding the the simulation, benchmark and modules setup are described.
The gained results - which have been obtained after simulating the benchmarks on the implemented interconnect-fabric models - are represented in Chapter 5. At the beginning of the chapter, a simulation example for both of the considered architectures is given (5.1).The distributed-memory based model and the shared-memory based model will be evaluated for each of the representative applications: banking (5.2), smart metering (5.3) as well as secure software update (5.4).
In Chapter 6 the conclusion about which model to choose for which application is shown (6.1). Finally, the future work (6.2) - where additional interesting communication architectures for evaluation as well as a further increase of the accuracy of the benchmark system and HW-models are pointed out - is discussed.

Chapter 2

Related Work

2.1 Communication Architecture Exploration

As listed in Section 1.3 one part of this work is to compare multiple different intercommunication-architectures. To be able to do this, first the classical designflow of System-on-Chips (SoCs) has to be reviewed to understand how HW is built in general and to know on which abstraction layer one should implement the model. This is done in Subsection 2.1.1.
Also the correct type and level of detail of the models has to be figured out in order to get meaningful results for the performance evaluation. This in turn allows to make correct communication architecture decisions and is presented in Subsection 2.1.2.
A typical environment for communication architecture exploration used in [WDL$^+$05] and in [SPD08] is outlined in Subsection 2.1.3. It builds the basis for this work's architecture explorations as well.
The common communication architectures which have to be considered are listed in Subsection 2.1.4. Their properties are discussed and a coherent decision on the used variants is shown.

2.1.1 Levels of Abstraction

The design-flow of SoCs - at least an idealized version - is presented in Figure 2.1. The general character and idea behind the typical HW-designflow is *divide and conquer* as it is in every abstraction-layer scheme. This means the starting point is at a very abstract level - the *functional model* layer - where the *application requirements* are expressed in a very high level model (e.g. *C/C++*) just fulfilling the functional requirements. This model does not consider any timing aspects of the model as it is written in a high-level programming language which has no component such as *Time* and therefore cannot specify "really controllable" concurrent processes[1]. [PD10]

[1] Threads allow real concurrency on multiprocessor-systems in the manner that multiple threads are really executed in parallel at the same time. It is though not controllable how the underlying OS selects these threads for execution. Therefore, it cannot be guaranteed that two specific threads are executed in real parallel.

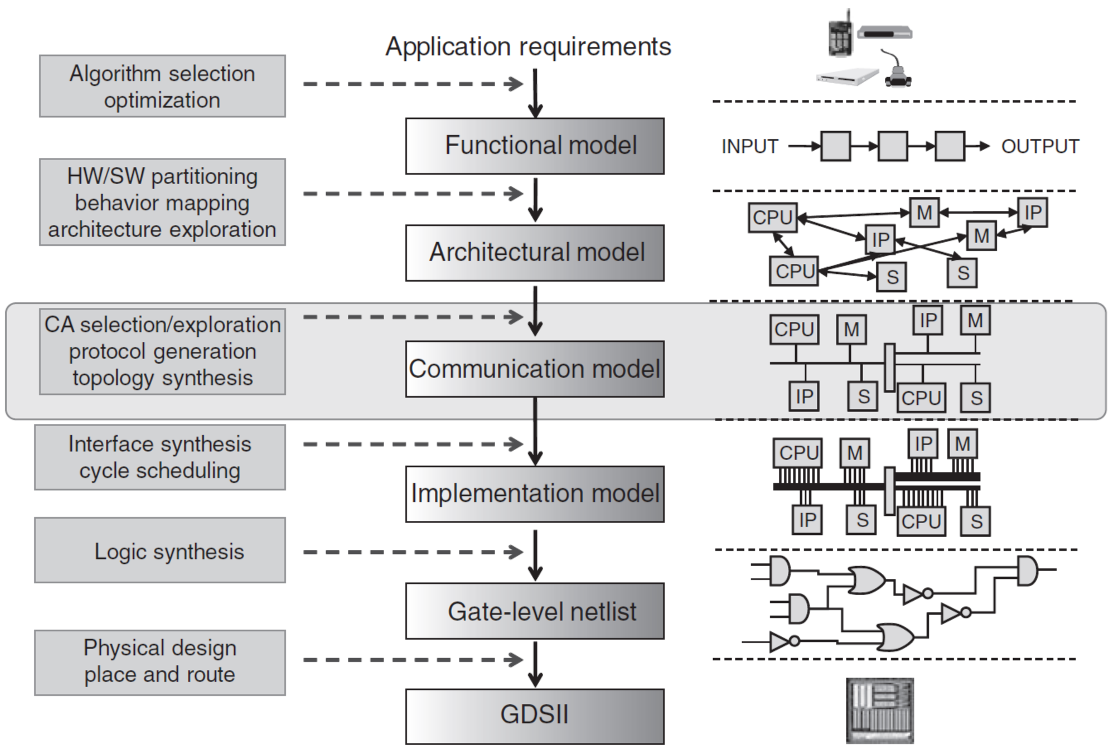

Application requirements
Algorithm selection optimization
Functional model
INPUT
OUTPUT
HW/SW partitioning behavior mapping architecture exploration
Architectural model
CPU
M
IP
S
CA selection/exploration protocol generation topology synthesis
Communication model
Interface synthesis cycle scheduling
Implementation model
Logic synthesis
Gate-level netlist
Physical design place and route
GDSII

Figure 2.1: Typical hardware designflow with various abstraction layers [PD10, page 5]

The *architecture model* layer is reached by an HW/SW partitioning of the functional model. This means, designers make a trade-off between which functional parts to put into HW and which to put into SW. Depending on timing-, power- and area-restrictions some parts are rather implemented in SW which leads to higher code size and computation time but smaller chip size and design-time, or to implement them in HW with its higher power-efficiency and performance but also higher design-time- and power- and chip-area-costs. The communication among these components, however, is still modeled in an abstract manner with some sort of message-passing. [PD10]

In the succeeding *communication model* these communication aspects are considered as specific communication architectures (e.g. bus ring, hierarchical bus, etc.). Assuming an architecture has been selected, further evaluations regarding the concrete topology with its detailed arbitration scheme, bus-width and -frequency are carried out. [PD10]

In a next step the components are specified further to describe their functionality in a cycle-accurate manner. Additionally, every interface is expanded to be pin-accurate, thus every single wire and connection among the components is declared. The result of this detailed description is the *implementation model* or better known as *Register Transfer Level (RTL) model*. [PD10]

Finally, these RTL model is used as an input for the completely tool-driven logic synthesis step - including the creation of the gate-level netlist files, floorplanning, place and route, etc. The resulting Graphic Data System II (GDSII) file is then used for tape-out, this means this file is sent to the fab (i.e. wafer factory) for fabrication of the final chip. [PD10]

2.1.2 Performance Estimation

Due to the evolution of SoCs' bottlenecks away from plain CPU power towards the communication architecture, it is crucial to perform evaluations that capture all the single aspects being responsible for these bottlenecks.
It is very important to include all delay-factors such as synchronization, marshalling-costs[2], the wire-propagation itself, cross-bridge transfers, etc. in order to gain realistic results that are quite close to its real performance. [PD10]

Static vs. Dynamic Performance Estimation

There are two common ways of performance estimation present all with different advantages and disadvantages, *estimation-based* (static) approaches and *simulation-based* (dynamic) approaches.

Static Performance Estimation: This method performs an analysis-based estimation usually consisting of closed form expressions describing the systems' performance. Typically, the single parameters describing e.g. the number of entities (so-called Processing Elements (PEs)) communicating with each other, the overhead introduced by the communication, etc. are adjustable. However, these static estimations rely on various assumptions modeled in a probabilistic manner, hardly consider non-deterministic traffic introduced by components on the bus, and of course they are unable to model dynamic delays correctly (arbitration, congestion, cache misses, buffer-overflows, etc.). [PD10]

Dynamic Performance Estimiation: This method uses simulated models to evaluate a systems' performance which allows to gain more precise timing values as also dynamic processes can be described. Of course, the effort to perform dynamic performance estimations is higher compared to static performance estimation. How much higher depends on the level of detail the model is based on. [PD10]

Dynamic Performance Estimation Models

There is a tradeoff between the level of detail of a model (and thus the accuracy of its results) and the simulation speed of the model. As depicted in Figure 2.2, various levels of detail for a communication model are possible [PD10].

On the one hand, each layer the level of details being captured rises when going from top down, resulting in slower simulation speeds on the other hand:

[2] Data bits introduced by the protocol - additional to the plain data - needed for a correct transmission of the data.

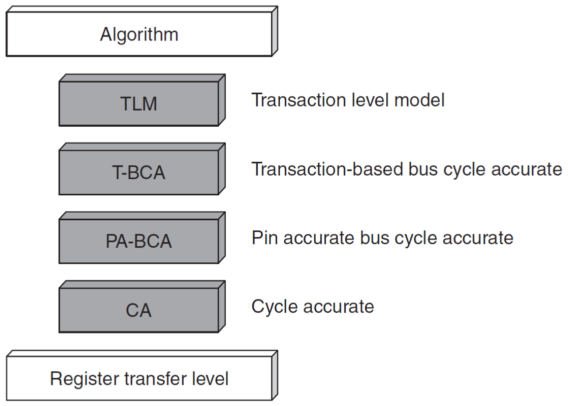


Figure 2.2: Abstractions of dynamic performance estimation models [PD10](page 112)

Cycle Accurate Models

This is the model with the highest accuracy and thus, the highest effort to implement. Everything is modeled in a pin-accurate, cycle accurate manner. That means, every single wire needed for e.g. a connection is modeled, the protocol specific delays that are introduced are captured, the components' calculations are scheduled at cycle boundaries. As this model is already quite close to the RTL model which is used for synthesis later on, such a model is often used for debugging and verification. Still, this model type is 10 to 100 times faster than RTL simulation because of the high-level language that is used. Due to its high effort of implementation, this layer does not suit well for an early architecture exploration. [PD10]

Pin-Accurate Bus Cycle Accurate Models

The Pin-Accurate Bus Cycle Accurate (PA-BCA) models are still implemented in a pin-accurate way but only partially cycle-accurate anymore. Thus, all connection-wires used for interconnection are modeled. However, cycle-accuracy now only holds for the bus (or connection) signals. Inside the components multiple instructions are grouped together and time is incremented on a multiple-cycle basis. This relaxed timing behavior leads to a faster simulation speed as time is not forwarded on a per-cycle basis anymore but also hardens debugging inside components as there is no cycle-accurate resolution anymore. Typical speedup of such models is in the range of 100 to 500 times faster than RTL. [PD10]

Transaction-Based Bus Cycle Accurate Models

As the PA-BCA model is still pin-accurate, the next step is the Transaction-based Bus Cycle Accurate (T-BCA) model where not only the computation inside a component is modeled in an abstract way (and thus is not generally cycle-accurate anymore) but also the connection of the components. This means the connections are not modeled pin-accurate anymore, leading to a generalization of the transfers. The various signals and wires of e.g. a bus connection are summarized to a *channel* and *transactions* on this channel. On this so-called *transaction level* convenience interfaces are present to simplify data read or write data communication events. This simple function calls are *read()* and *write()* with address, data and control infor-

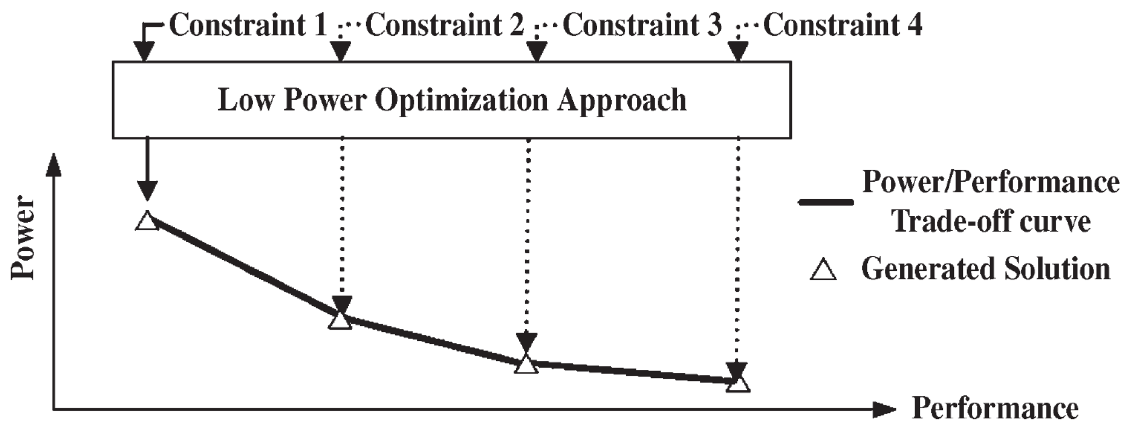


Figure 2.3: Basic concept of pseudoparallel bus architecture exploration method at system level [CCL09, page 3]

mation as parameters. Obviously, this simplifies the communication process a lot, leading to an up to 1000 times faster simulation speed. The lack of visibility of the signal transitions is the model's drawback. [PD10]

Transaction Level Models

Transaction Level Modeling (TLM) is a very high-level approach without any pins or signals being modeled. Transactions are done similarly to the PA-BCA model with read() and write() functions but, as this model is protocol-independent, without control information. Only address and data is declared which then leads to an instantaneous data transfer, timing delays for computations are optional, synchronization is achieved via *wait()* statements. Without any timing notations of the protocol, such models are unsuitable for communication architecture exploration. However, there are also approaches to include bus protocol timing details even on TLM level. Examples are [SD06] and [ABR$^+$03]. Of course, some inaccuracies arise due to such statical estimations of bus arbitration, contention and pipeline delays. [PD10]

Static Performance Estimation based Archictecture Exploration

In [CCL09] an example of an analysis-based architecture performance exploration is discussed. A so-called pseudoparallel method for bus architecture exploration at the system level is demonstrated. The goal is to find interesting variants of a bus-based communication solution regarding its power- and performance-characteristics. The pseudo-parallel structure of their algorithm allows them to investigate interesting portions of design space faster, still considering key constraints such as a maximum delay in terms of cycles. An overview about the basic idea is given in Figure 2.3.

Their algorithm also requires little computation cost on the estimation of dynamic factors such as bus conflicts since a probabilistic metaalgorithm is used. To generate the various variants of the bus-based communication architecture, the following strategies are used:

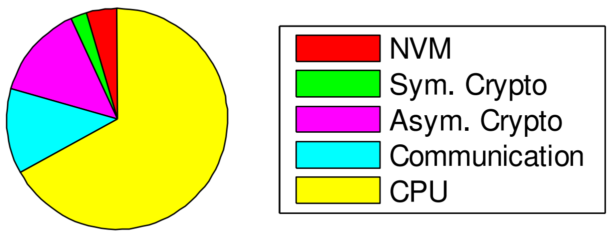


Figure 2.4: Timeslots of various HW modules at a passport readout application [Pöl12, page 77]

- Merge two buses into one signal bus.
- Split one bus into two seperated buses.
- Move one component to another bus.
- Exchange the locations of two components on different buses.
- Change bus parameter settings.

Dynamic Performance Estimation based Architecture Exploration

An interesting simulation-based performance estimation approach is presented in [Pöl12]. Different smartcard applications from different domains such as passport or banking, are evaluated regarding their hardware requirements. Similarities among one domain's applications are investigated which should allow to draw inferences from existing applications' requirements to new ones.
The simulation model is based on the execution of application specific models (i.e. benchmarks) that are executed on TLM models representing the smartcards' HW components. Here, not the communication among the components with its time-critical sections is considered but a functional implementation of the various components is modelled to allow a detailed logging of each components behavior. To allow a detailed analysis of the single application's behavior, the partition of the simulation-based logging is as follows:

- Memory accesses and its associated times.
- Stack-activities of the microprocessor.
- Activities of each HW module.

An example result is presented in Figure 2.4 containing the time consumption of the dedicated HW modules such as Non-Volatile-Memory (NVM), symetric as well as asymetric cryptography performed by co-processors and the CPU itself.

Another interesting dynamic performance estimation based benchmark approach for smartcard systems is presented in [WWRL08]. Here, a smartcard's processor being designed for *Java* applications is benchmarked. Their *Java Card System* consists of the 16-bit processor, the Java Virtual Machine (JVM) running on it and the applications running inside the JVM. Their processor is based on a general-purpose processor - the Intel 80251 - and they try to improve the processors' performance for specific smartcard

applications. They use benchmark applications to identify the systems' bottlenecks.
"The key idea is to select representative programs from each cluster of similar programs so that the whole application space is represented by a subset of programs." [WWRL08, page 18]
After the identification of the most representative applications, they evaluate the bytecode usage of these programs to identify the composition of the programs. This allows the detection of the bottlenecks - for the *Dhrystone* benchmark this is the *array_store* operation (updating the NVM) for example.

In [PPM+07] a communication architecture is evaluated towards its performance (latency and throughput). Also user-level benchmark applications are used to evaluate the systems' performance. The communication infrastructure they evaluate is a 4x4 64-bit Peripheral Component Interconnect (PCI)- Express network architecture.
Their benchmark approach is to further abstract the benchmark system and simply use so-called *Traffic Generators* to generate the abstract packet-traces which are sent across their communication network.

2.1.3 Simulation Environment

As already elaborated in Subsection 2.1.1 the layer that fits best for performing communication architecture exploration is the communication layer. This layer is characterized by a detailed description of not only which components are connected to each other but also describes the explicit protocol in use. Thus, the protocol-related delays[3] are also considered.
As described in [WDL+05] an ideal environment setup is shown in Figure 2.5, suiting perfectly for this layer of abstraction. This means, it allows all aspects of the communication architecture and its underlying protocol to be modelled correctly while the simulation speed is also kept on an acceptable level.

Language for Instruction Set Architecture (LISA)

The Architecture Description Language (ADL) Language for Instruction Set Architecture (LISA) is used to describe the processor's instruction set architecture on system-level. This allows to generate the Instruction Set Simulator (ISS) and other tools used to simulate program code execution on a processor.
The basic principle of the LISA language is to use two types of information to describe the behavior of a processor. Firstly, the storage elements which describe the actual state of the processor, and secondly, a detailed bit-true behavior description of the processors' instructions - the transition functions between two states.
LISA models use so-called *resources* to hold the actual state of the architecture - the values of registers, memories and pipelines - by saving the stored data value together with an additional information describing the resources availability for operations.
The *operations* are used to model the second part of the language - the state transitions. As these operations build the basic objects of the language and are used to delegate the system

[3] Resulting from the overhead introduced by the protocol to enable an error-free transmission of the data in its correct sequence.

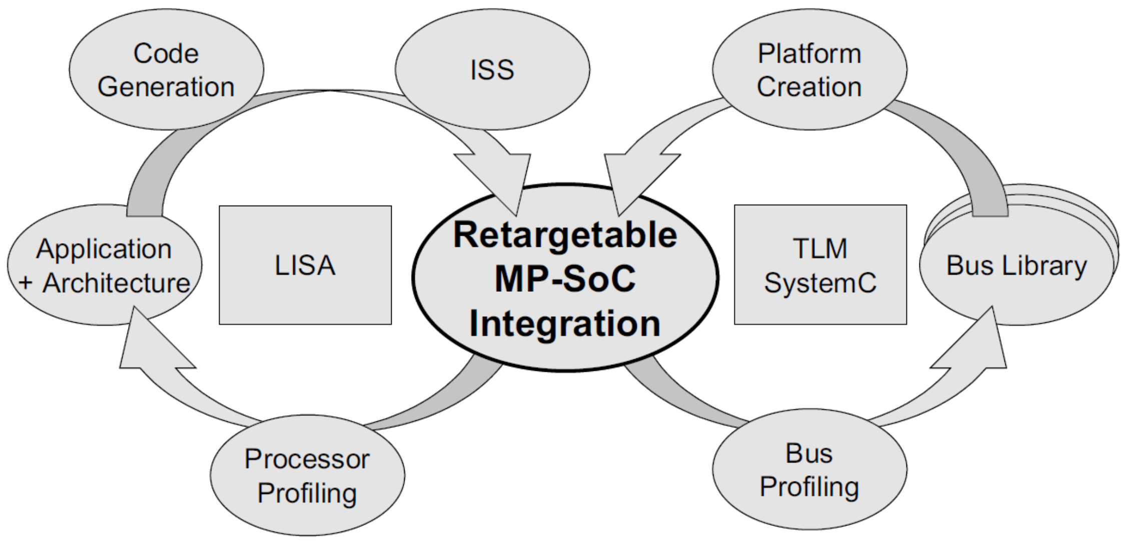


Figure 2.5: Simulation environment to evaluate communication architectures used in [WDL+05, page 3]

from one state to another while considering a lot of parameters, their definition contains a lot of information. Operation definitions include an operation behavior, instruction-set information and a timing description in order to be able to go from one state of the system to another in a structured way.
There are *atomic* - unsuspendable operations, not further split - as well as combined operations which consist of multiple sub-operations, leading to an operation hierarchy.
With this information, actual state and available operations - the model of a processor can be fully specified. Depending on the needed accuracy, a designer may describe the operations as single instructions (instruction-based model) or as a composition of actions being performed between single cycles (cycle-based model). [HML02]
ARMs' SystemC TLM2 compliant Instruction Accurate (IA) ISSs are called *Fastmodels*. There exist Cycle-Accurate (CA) ISSs for ARM processors as well. The latter are only used for HW validation and due to their very slow simulation speed often directly the RTL model is simulated. As this work's intention is to perform an architecture exploration its focus is on the IA fastmodels. The models are generated and also configured via the *System Canvas* tooling which uses LISA processor models.

SystemC

SystemC is a IEEE-standard [Ini11] and a C++ class library that can be used to build cycle-accurate models of HW architecture, SW algorithms as well as interfaces for the SoC design. There are various advantages for using SystemC as modeling language: digital HW designers are often already familiar with the C/C++ language, thus the effort for learning how to use SystemC is very low. As it is a C++ extension, all commonly used C++ development tools can be used too. Therefore, SystemC enables the designers to create very accurate models very fast, resulting in an executable description of the final system, which is basically a C++ program like others running on a normal PC. [I+01]

C and C++ is widely used for system descriptions because of their nature providing the necessary control and data abstractions. The choice to use C++ as the basis for SystemC is due to its object-oriented concept which allows simple extensions of the original language through classes. Therefore, necessary constructs for modelling architectures such as HW timings, concurrency or reactive behavior (i.e. so-called *events*) can be added easily. Trying this with the C language would result in proprietary languages which could never become a standard. [I^+01]
In SystemC, structural description of the HW component is done in *modules* whereas behavioral description of the component is done in *processes*. Ports are used to define an entities' interface to other components. These ports can be connected via *channels*. Thus, exchanges of functional as well as communication related parts is very easy. Also profiling is simplified and can be performed on very specific parts. [I^+06]
Although, already SystemC 2.3 [Ini11] is available by the time this work has been written, SystemC 2.2 [I^+06] is still widely used. The newly introduced features of SystemC 2.3 can be summarized as follows [Ini11, pp. 585]:

New process control functions:
: Processes can be suspended, resumed, reset or killed by themselves or other processes.

Unifying of thread and clocked thread processes:
: All types of processes can now have any number of synchronous or asynchronous resets.

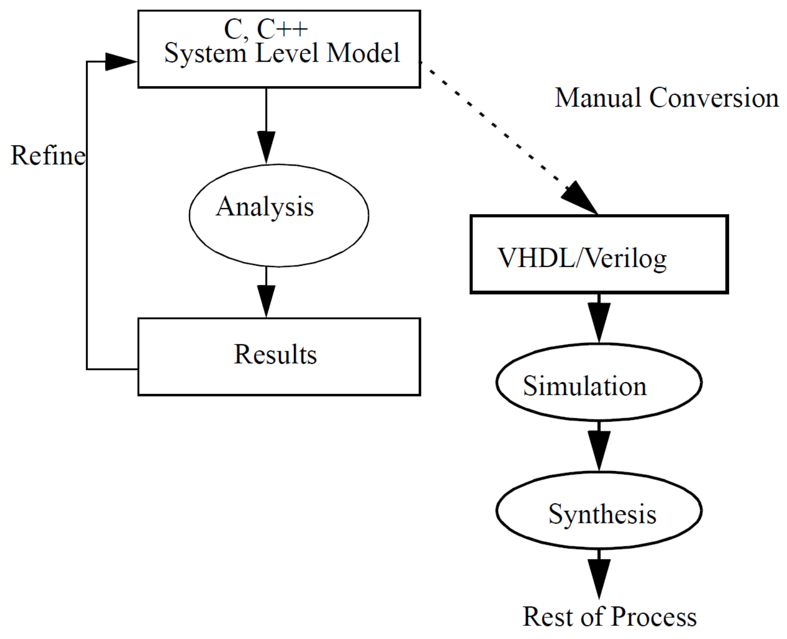
Pause of simulation:
: Now a pause of simulation is possible and retrieving the actual state of simulation is also available.

Events naming convention:
: A hierarchical naming convention for events in the same way as *sc_objects* has been introduced.

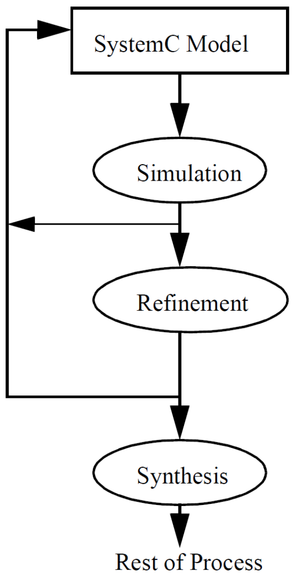
Minor changes:
: new vector class *sc_vector*, *sc_mutex* and *sc_semaphore* now derived from *sc_object* (allows a dynamical instantiation), change of *boost* to *sc_boost*, etc.

Figure 2.6 compares the methodologies for HW design descriptions before and after the introduction of SystemC. It can be seen, that the time-consuming, error-related step of the manual conversion from a functional description model in C++ for example, to a Hardware Description Language (HDL) such as *Verilog* or *Very high-speed Hardware Description Language (VHDL)* can be skipped. If necessary, SystemC can be used to model the system from system-level down to RTL in clear refinement steps. [I^+01]

In [CGO01] a specification language on system level called *SpecC* [GZD^+00] is used. SpecC is a super-set of C that extends American National Standards Institute (ANSI)-C and therefore allows the reuse of already existing C descriptions - either algorithmic or behavioral. It contains also all features neededfor a detailed system-level design specification.



(a) The HW design methodology without SystemC shown in [I+01, page 4]



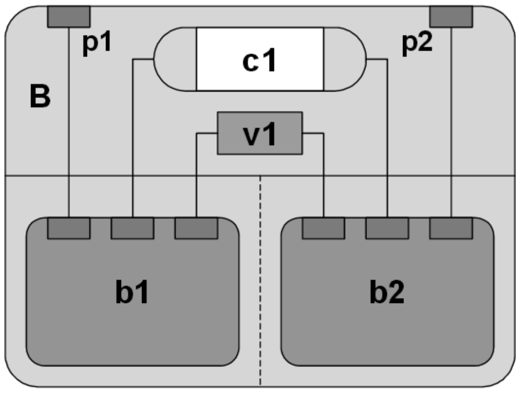
(b) The HW design methodology with SystemC shown in [I+01, page 6]

Figure 2.6: Comparison of the HW design methodologies with and without SystemC

In SpecC, three basic components exist: the behavioral description is located inside *behavior*-blocks, *channels* are used to connect various components to each other, and *interfaces* which can be implemented by channels or ports. A behavior definition consists of multiple ports, local variables, instantiations and methods - inclusive a *main* method which is mandatory. [DGKO02]

An example of a structural hierarchy in SpecC can be seen in Figure 2.7, where a behavior *B* has two ports *p1* and *p2* for external interaction and two child behaviors *b1* and *b2*. These are connected with each other via a channel *c1* as well as shared-variable *v1*. The interface *I1* is implemented by C1 and is also used to define one of the ports of B1. The keyword *par* is used to state the parallel execution of the two sub-behaviors. [DGKO02]

In [CVG03] a comparison between SpecC and SystemC is performed. They critisize that SystemC is C++ based which leads to a harder timing profiling due to the library burden. The harder merging of parallel processes during the refinement steps towards lower abstraction is also listed. Also the lack of static scheduling which is used to determine an explicitly modeled execution sequence during architecture exploration, as well as the lack of a hierarchical behavioral description is mentioned. For some of SystemC's disadvantages workarounds exist and on the whole these System Description Languages (SDLs) are quite similar.



```
interface I1
{
  bit[63:0] Read(void);
  void Write(bit[63:0]);
};

channel C1 implements I1;

behavior B1(in int, I1, out int);

behavior B(in int p1, out int p2)
{
  int v1;
  C1  c1;
  B1  b1(p1, c1, v1),
      b2(v1, c1, p2);

  void main(void)
  { par { b1.main();
          b2.main();
        }
   }
};
```

Figure 2.7: Structural hierarchy of SpecC as outlined in [DGKO02, page 8]

Transaction Level Modeling

The IEEE-standard TLM[Ayn09] - which is part of the SystemC library since SystemC version 2.3. - is used to model the various attached HW components as well as the communication architecture. "TLM is a transaction-based modeling approach founded on high-level programming languages such as SystemC. It highlights the concept of separating communication from computation within a system." [G+05]
This means, in TLM the various components - modeled as a set of concurrent processes - are connected to each other via abstract channels. The communication is enabled through so-called *transactions* over these abstract channels. Thus, the functionality of the components is modeled inside their processes and the communication is modeled via the abstract channel interfaces. These abstract channels allow a very high-level implementation without any underlying protocol as well as a detailed description of the protocol-related delays, depending on the needed level of detail. The final TLM designflow is shown in Figure 2.8 [G+05].

For TLM2 there exists the Loosely Timed (LT) and the Approximately-Timed (AT) coding style[4] The first is implemented by the so-called *b_transport()* interface function, thus the blocking transport - which has two timing points, the start and the end of the transaction - allows a somewhat coarse grain synchronization at specific points (the return-

[4] Actually, there exists a third one, the *Untimed* coding style. Here, no compulsory notion of simulation time exists. However, this scheme is only used for functional verification without considering any time-related aspects.

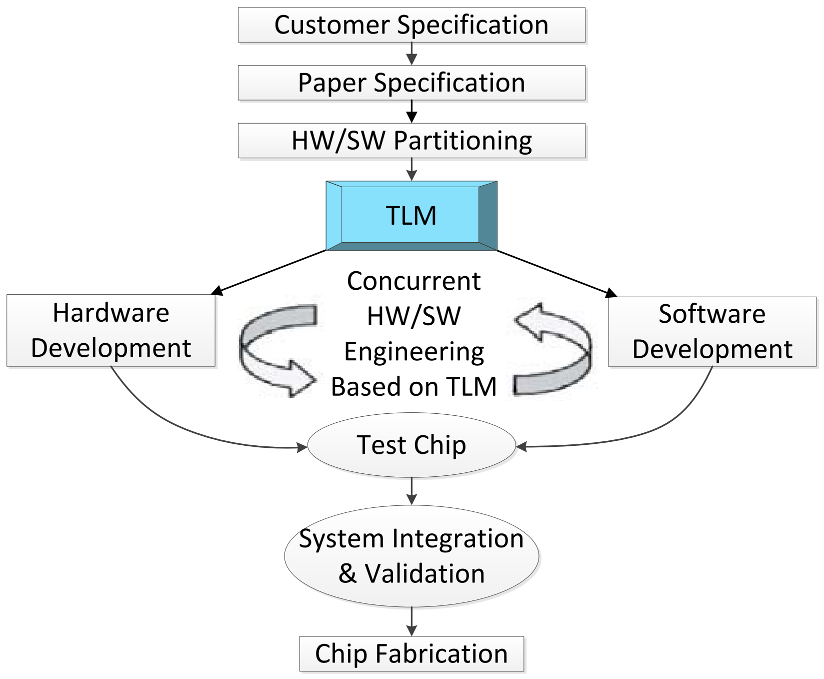


Figure 2.8: TLM design methodology overview depicted in [G+05, page 16]

ing of the function). Whereas the AT coding style is implemented by the *nb_transport()* functions, thus the non-blocking transport - which has four timing points, the start and the end of the request as well as the response.
However, it is also possible to switch between LT and AT modelling. The idea thereof is to run fast through uninteresting parts of the simulation and switch to AT for more interesting parts. [Ayn09]
As depicted in [Ayn09, page 11] the LT models are totally applicable for HW architectural analysis. For a HW performance verification the AT or even the CA models have to be used.

The TLM methodology is widely recognized and not only available for SystemC. In [GB11] the usage of TLM in *SystemVerilog* - an extension of the HDL *Verilog* to model on the system level - is shown. A translation of the TLM 2.0 definition - which is written in C++ - has been integrated into Universal Verification Methodology (UVM). As C++ allows multiple inheritance and SystemVerilog does not and due to other singularities of the C++ language definition, some parts had to be adopted but in general the same functionality regarding TLM is now available for SystemVerilog.
An implementation example of the typical TLM interface functions for SystemC translated into SystemVerilog can be seen in Figure 2.9. As the parameters are passed by reference, an ampersand (&) is used for SystemC whereas the *ref* keyword is used for SystemVerilog.

```
void b_transport(TRANS& trans,
                 sc_core::sc_time& t);

tlm_sync_enum nb_transport_fw(TRANS& trans,
                              PHASE& phase,
                              sc_core::sc_time& t);

tlm_sync_enum nb_transport_bw(TRANS& trans,
                              PHASE& phase,
                              sc_core::sc_time& t);
```

(a) Transport prototypes of the TLM2 methodology for SystemC [GB11, page 2]

```
task b_transport(T t, uvm_tlm_time delay);

function uvm_tlm_sync_e
    nb_transport_fw(T t, ref P p,
                    input uvm_tlm_time delay);

function uvm_tlm_sync_e
    nb_transport_bw(T t, ref P p,
                    input uvm_tlm_time delay);
```

(b) Transport prototypes of the TLM2 methodology for SystemVerilog [GB11, page 2]

Figure 2.9: TLM2 Transport Interface Protoypes

[RSG+04] investigates the similarities and differences of SystemC and SystemVerilog. The authors come to the conclusion, that on the whole both languages allow an implementation on the same level of abstraction, reaching from the very abstract system-level down to RTL if necessary.
Typically, SystemC is used to start from top-down allowing a good HW/SW codesign as also SW can be integrated easily since it is C++ based. Whereas, SystemVerilog is used from bottom-up, started with a Verilog description and going up in the abstraction level to build fast verification models.

In [SPD08] they evaluate the performance of various architectures with different general purpose processors by cross- compiling application code (C code) and execute it on the proper ISS to extract the performance trace. Afterwards they put the results together with the cross-compiled binaries into a code annotator which produces annotated code that allows to extract the running time of individual code segments. The timing values are then used to refine the SystemC models. An overview about this setup can be seen in Figure 2.10.

Their used setup with the Keil ARM development tool, the virtual hardware (ISS) with its embedded profiler (cycle counter) is very similar to this work's performance evaluation setup. In contrast to this work's evaluation of different communication architectures' performance for different applications, they evaluated the influence of different processors (ARM9, ARM7 and Xilinx' Microblaze) being responsible for different parts of the

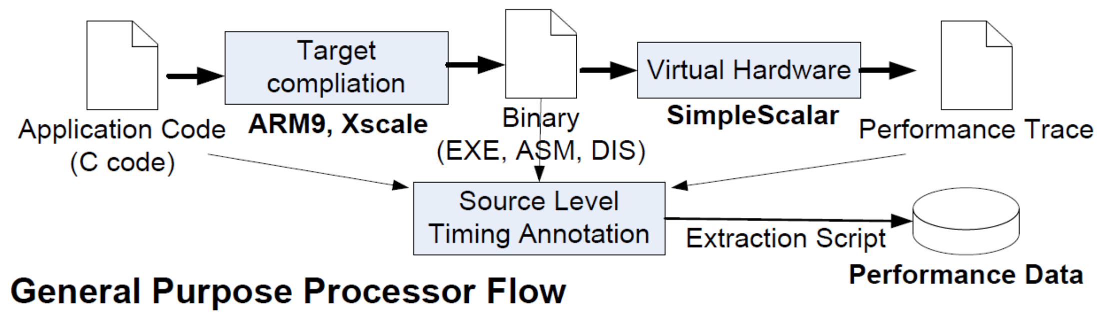


Figure 2.10: Performance evaluation setup for general purpose processors [SPD08, page 4]

physical and data-link-layer of the Universal Mobile Telecommunications System (UMTS) protocol.

2.1.4 Common Communication Architectures

The communication architectures exposed in [Lia04] can be summerized to the following basis-structures:

Fixed point-to-point connection
: The components are connected directly among each other via dedicated wires. Due to the fact that one connection is only used by the two components being connected, no bottleneck arises with this communication scheme. The disadvantage is its limited flexibility since the connections have to be decided during design phase. In small chip designs the limited flexibility is negligible. [Lia04]

Bus-based connection
: In a bus-based connection scheme the components are connected via long bus wires which are shared among each other - often in a timely manner. The decision which entity gains access in the case multiple components try to access the bus at the same time, is often done in a priority fashion. The so-called arbiter then selects the module with highest priority to perform its request while the others are declined and will try again later. This leads to a highly flexible design where additional components can be added very easily but at the same time having the drawback of only one connection between two components can be active at the same time. [Lia04]

Network-on-Chip (NoC)
: In an NoC the components are connected to each other via a network of switches building a mesh-structure where each entity is connected to everyone. Therefore, packets with an address and some data are sent through this network being routed by this previously mentioned switch-fabric. This solution is highly scalable allowing hundreds or even thousands of components to be connected to each other. The disadvantage here is the introduced complexity of the switching-fabric with its dynamic routing, buffering, flow control etc. Thus, this solution is a candidate for future SoCs. [Lia04]

2.1.5 Summary

In order to be able to perform a communication architecture exploration, first of all the correct layer of abstraction to perform the desired measurements on has to be detected. At the beginning of this chapter - introducing the levels of abstraction in the HW designflow - it is apparent that the communication abstraction layer is adequate for the communication architecture exploration. On this layer, enough details of the communication among the components are captured while at the same time the design time is kept on an acceptable level.

The next subsection reveals differences of static and dynamic performance estimation approaches. This work's focus is kept on the latter approach since modern SoCs' bottlenecks are nearly all related to dynamic processes as well as its higher overall accuracy over the static performance estimation. However, for the sake of completeness also a static performance estimation approach is shown.
The static approach illustrated in [CCL09] proposes an interesting algorithm for a very fast communication architecture exploration, finding the solution with best power / performance characteristics. However, only bus-based communication is considered. Thus, their algorithm is adequate in a second design step, when the decision which communication architecture to use has already been made. It can be seen as a sort of finetuning step to find the best arrangement of the single components especially inside one sub-system.
Thus, to allow an accurate performance evaluation and also consider dynamic processes, simulation-based performance estimation models are supposed to be applied. The level of detail of this models is a tradeoff between accuracy and simulation time. For communication architecture exploration the T-BCA model would fit best. For very early architecture decisions also a very detailed TLM level model can be used when dynamic processes such as bus- arbitration or -congestion delays are considered - at least in a statistical manner.
An interesting performance evaluation approach is shown in [Pöl12]. As can be seen in Figure 2.4, the processing as well as the communication part of overall application duration is quite high. Therefore, this works' communication evaluation is of high interest in order to evaluate the bottleneck of smartcard applications.
In contrast to the dynamic performance approach shown in [WWRL08], this work's intention is not to benchmark the sole processor power or in-system components regarding single applications but to investigate the influence of different communication architectures for specific applications. However, their idea of using specific benchmarks to represent most of the applications later in use on the smartcard is very interesting.
The simple abstraction - using *Traffic Generators* - of the second dynamic performance approach used in [PPM$^+$07] is, however not applicable for this work's evaluations since the selection of the best communication infrastructure depending on specific applications requires also additional modelling. For example, specific delays depending on the content could arise inside the application, which is not addressable by this traffic generator.

A further step towards a practical, accurate and easily configurable simulation environment setup is done in the next subsection with the discussion of the simulation environment used in [WDL$^+$05]. The main idea in [WDL$^+$05] is to constantly refine the LISA-based processor model as well as the TLM-based HW component descriptions towards a more

accurate communication model to allow a more detailed analysis.
Multiple bus-based communication architectures are evaluated for MPSoCs whereas this work's focus lies on finding an appropriate communication architecture (either bus-based or point-to-point connection) for specific smartcard applications.
All components of their setup such as the LISA-based ISS and the modelling-languages *SystemC* and TLM are described in detail as well in this Subsection.
Also alternatives are presented, for example the *SpecC* language used in [CGO01], or the usage of TLM in *System Verilog* by [GB11].
[RSG$^+$04] investigates the similarities and differences of SystemC and SystemVerilog and comes to the conclusion that SystemVerilog is used from bottom-up, started with a Verilog description and going up in the abstraction level to build fast verification models.
[SPD08] performs architecture evaluation with a setup consisting of cross-compiled C-code to be executed on an ISS. They used the results of the ISS to evaluate the differing runtimes of the functional blocks. As this work's evaluation is about the communication infrastructure between two μCs, not the timing values of the ISS runtime will be of interest (does not change for the different communication architectures) but the additional cycles introduced by the interconnection fabric which will be considered in the SystemC models. This is an interesting approach for executing specific applications on the sub-systems' CPU.

In the last subsection, all common communication architectures are presented. According to [Lia04] these are the point-to-point connection, the bus-based connection as well as NoC.
Due to the fixed number of sub-systems that have to be connected - only two - the last one is not of interest for this work's evaluations. Considering the target platform, the most promising communication architectures are a fixed point-to-point as well as a bus-based connection solution. Thus covering both, a low-area as well as a high-performance solution.

2.2 Distributed-Memory based Models

The target platform describes a MIMD system as already discussed in Subsection 1.1.4. The classification of these systems is stated in Figure 1.3. Most of such systems can be classified as either *distributed memory* or *shared memory* based. Thus, the possible communication architectures are also split up into these groups to be better comparable.
In general it can be said, that distributed memory based approaches contain some point-to-point connection between the two sub-systems and have a message-passing communication scheme. Thus, they exchange messages with some specific protocol to communicate with each other.
Approaches of each classification-group have similar properties therefore, they will be discussed seperately to allow a comparison among each category's candidates. A decision on which approach to use for the target platform is done in Chapter 3 while a quantitative comparison between the selected approaches is then performed in Chapter 5.

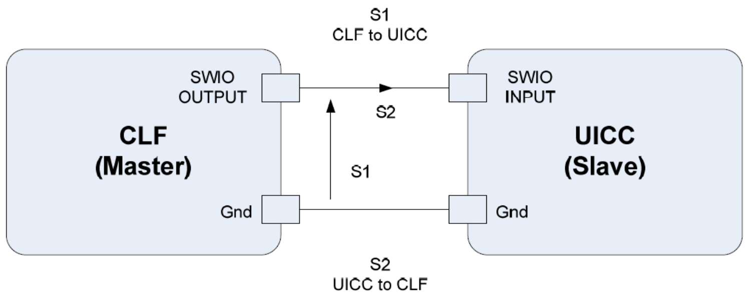


Figure 2.11: Data transmission of the two data links S1 and S2 [ETS08, page 10]

2.2.1 Single Wire Protocol

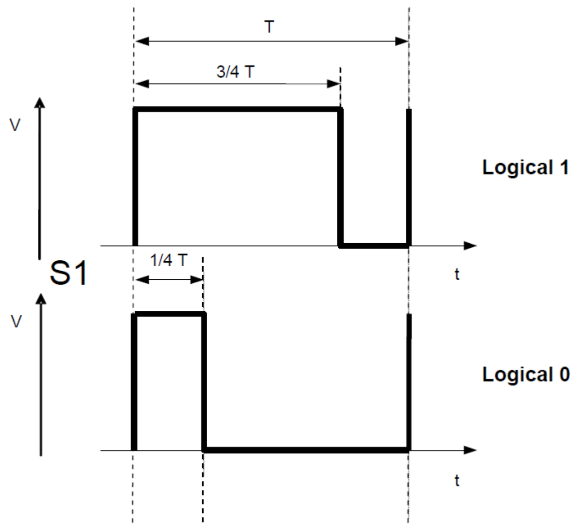
The Single Wire Protocol (SWP) is a protocol normally used in mobile devices to connect the Contact-Less-Frontend (CLF) with the UICC in order to establish a contactless communication between two devices' CLF interface. Therefore, the protocol specifies a contact-based communication used in contact-less communication between e.g. a smart-card and its reader.
More precisely, the protocol defines a master(CLF) - slave(UICC) communication method which is full-duplex, i.e. data can be sent in both directions at the same time. This is achieved by using the voltage domain for data to be sent from the master to the slave and the current domain for data to be sent from the slave to the master as described in Figure 2.11.
As the CLF gets powered passively over the electromagnetic field, this unit is the communication-master since it powers the UICC with its received energy. So, the protocol specifies a master-slave tranmission-behavior although the slave is also able to initiate a communication indirectly through the activation of the interface. This is done either via a *RESUME* command when in *SUSPENDED* state or via using the *ACTIVATE INTERFACE* command when in *DEACTIVATED* state as described in [ETS08].

To describe the data transaction in detail, a closer look on the physical layer (see 2.2.1) as well as the data link layer (see 2.2.1) has to be taken. The former defines the transaction of a particular bit over the single wire and the latter is used to describe how individual bits are grouped together in order to achieve a reliable and error-resistant communication.

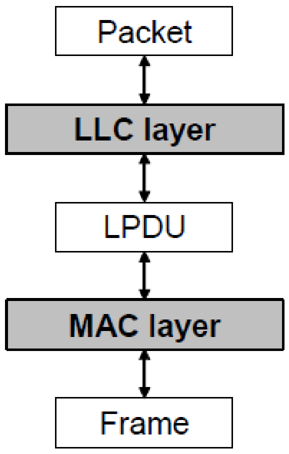
Physical Layer

The logical 1 is specified having a nominal duration of state H (high) for $0.75 * T$ and the logical 0 is specified having a nominal duration of state H for $0.25 * T$ as it can be seen in Figure 2.12a. The graphic shows the example for the signal *S1* (voltage domain) but this is equivalent to the definition of the signal *S2* (current domain), using the current thresholds instead of the voltage ones.
Therefore, both bit value possibilities are specified having a bit-duration of T, which is declared to be between $1\mu s$ (Minimum) and $5\mu s$ (Maximum)[5]. [ETS08]

[5] Actually, today's SWP implementations often already use smaller bit-durations of $0.5\mu s$ and less.



(a) Bitcoding specification for a logical 1 and a logical 0 [ETS08, page 20]



(b) Overview of the data link layer specification in SWP [ETS08, page 24]

Figure 2.12: SWP bitcoding specification (2.12a) as well as the data link layer setup (2.12b) [ETS08]

Data Link Layer

The data link layer is structured as described in Figure 2.12b. Thus, the data link layer consists of the Link Protocol Data Units (LPDU) which itself can be subdivided into two layers, namely [ETS08]:

Medium Access Control (MAC)
: As illustrated in Figure 2.13 this layer is responsible for the framing, thus the correct (error-)detection of frames. With zero- bitstuffing the unambigious detection of Start of Frame (SOF) and End of Frame (EOF) is enabled, while the CRC allows a proper error-detection inside frames.

Logical Link Control (LLC)
: The responsibilities of this layer include the error management as well as the flow control. A receiver-buffer overflow detection is also included in the error management capabilities as many others and allow a fast retransmission of frames in case of an error. The flow control is responsible for observing the correct sequence of the data packets. An example transmission where data is sent back from the slave to the master in a piggybacking fashion is shown in Figure 2.14.

The layout of a frame on MAC layer showing the payload and CRC being sorrounded by the SOF and EOF is shown in Figure 2.13. Bitstuffing is applied to the data between SOF and EOF to prevent an occurance of exactly the same bit-pattern which would lead to an incorrect SOF or EOF detection.

The payload now contains LLC layer data that is again divided into several parts which are not considered further. What can be seen in Figure 2.14 is that the data is exchanged

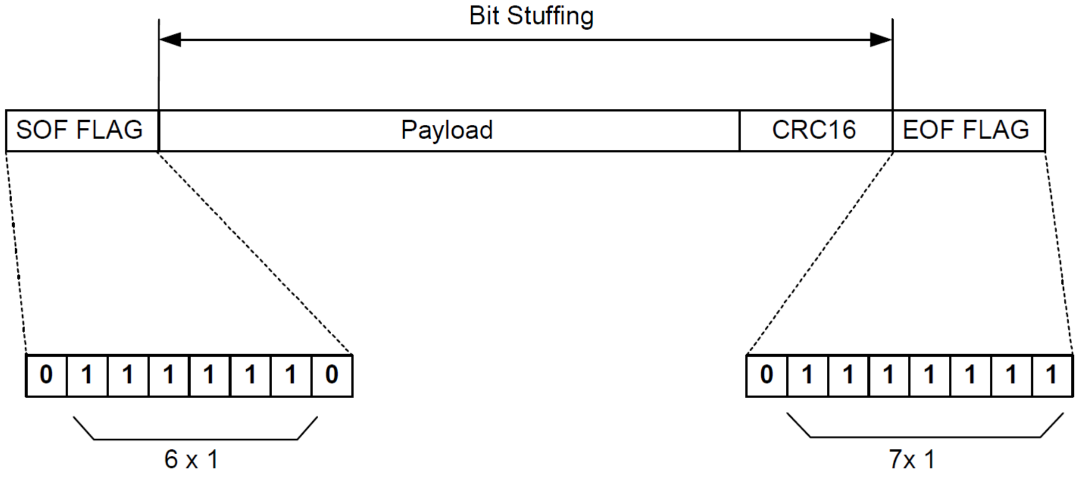


Figure 2.13: Frame layout of the SWP protocol [ETS08, page 25]

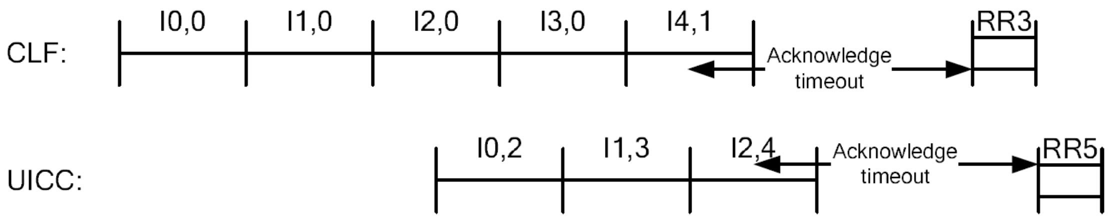


Figure 2.14: Example transaction of an LLC-based SWP transmission [ETS08, page 36]

via so-called (I)nformation frames. These are numbered according to the actual transmit-frame-number and the received-frame-number. Starting with I0,0 the master increments the first number for every new frame being transmitted. The master is allowed to keep sending frames as long as it stays within a so-called window of e.g. 4 frames. During frame I2,0 the slave starts with its own frame responses and implicit acknowlegments.
After that the master's window has extended to e.g. $2 + 4 = 6$ and also continues sending its frames with an incremented receive number I4,1. The buffer-overflow detection is included in the final acknowledge which is here the RR (Receive Ready). In case of an overflow this would be RNR (Receive Not Ready).

Transfer Rate

According to [ETS08], the bit-duration is defined to be between 1 μs and 5 μs. Although, nowadays the bit-duration goes towards less than 1 μs thus, leading to a transfer rate of 1 $Mbit/s$ and more.
However, these values do not describe plain data bits as the marshalling-cost overhead has to be considered. For SWP these marshalling costs sum up to about 5 header *bytes*

for a maximum of 29 data *bytes*[6] for the physical as well as data link layer [ETS08]. Any higher layers - which are optional and highly application dependent - of course introduce a further overhead.

2.2.2 Serial Peripheral Interface

The Serial Peripheral Interface (SPI) is a de-facto standard implementing a synchronous serial data link operating in a full-duplex way. It is synchronous as a clock is used (Serial Clock (SCLK)), serial as only one wire for each direction is used to transmit the data (Master Output, Slave Input (MOSI) and Master Input, Slave Output (MISO)) and full-duplex as data can be sent in both directions at the same time (one wire for each direction). It is a de-facto standard since many parts of the protocol originally invented by *Motorola* are not precisely defined. These parts are then specified by the particular manufacturer using this type of connection.
SPI can be used to connect a master with multiple slaves as depicted in Figure 2.15a, but it can also be used in a reduced variant connecting only one master with one slave, as it can be seen in Figure 2.15b. The latter variant is the one which would be interesting for our target platform. Obviously, for each additional slave that has to be connected, one additional signal line (Slave Select (SS)) is added. [byt]

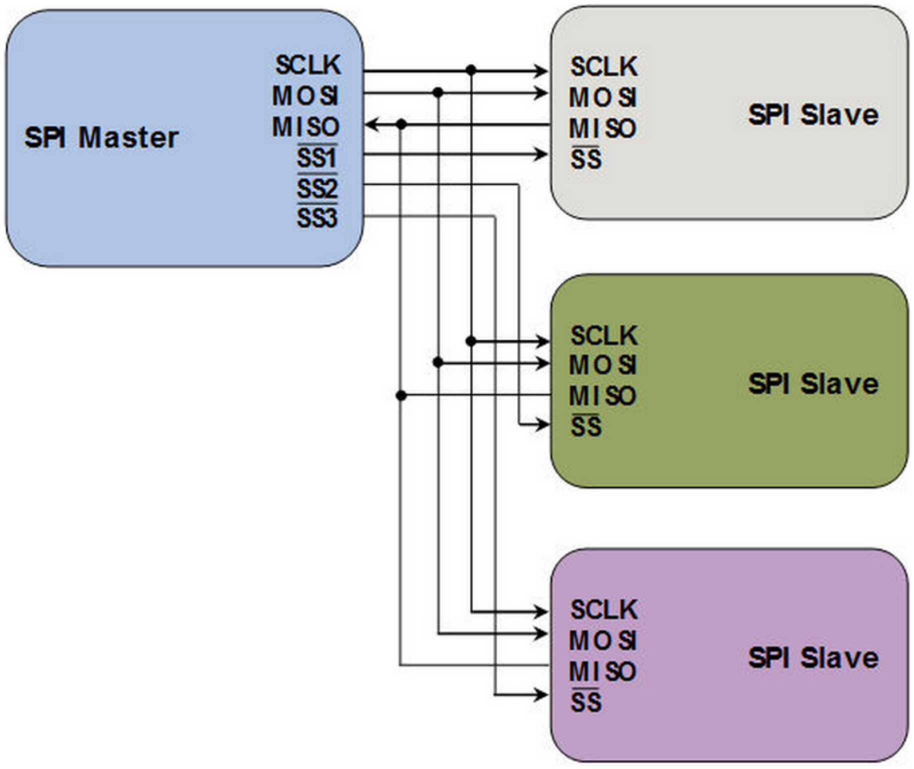
Physical Layer

From the physical layer's view there are four wires used to connect the components [byt]:

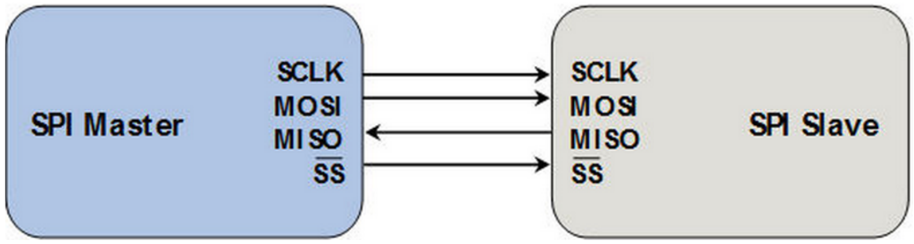
- SCLK - a clock signal sent from the master to (all) the slave(s).
- SS - slave select signal used by master to select a specific slave during the communication.
- MOSI - the master out-slave in line used to send data from master to slave.
- MISO - the master in-slave out line used to send data from slave to master.

When the master wants to send data to the slave, it selects the slave it wants to communicate with by pulling the SS line low, keeping it low until the end of the communication, setting a clock frequency for the SCLK line the master and the slave supports, putting data on the MOSI wire and sampling data from the MISO signal. An example communication (with $mode = 0$) can be seen in Figure 2.16.
Depending on which one of the four *modes* is used, data is toggled either on the falling edge and sampled on the rising edge or vice versa[7]. The four modes are illustrated in Figure 2.17, showing all combinations of Clock POLarity (CPOL) and Clock PHAse (CPHA). These are the interface's specifications, all higher layer protocols set up on top to guarantee data reception and enable any flow-control management is application specific and defined by the manufacturer. Since smartcard applications are typically used in a security-relevant environment, this overhead has to be considered in the evaluation and thus, it is discussed in Subsection 2.2.2. [byt]

[6] 1 byte for the SOF, 1 byte for the EOF, 2 bytes for the CRC and 1 byte for the LLC-header. The single bits added by the bitstuffing are not considered here.
[7] Conforms to two possibilities, together with the SCLK's polarity this forms four modes.



(a) SPI architecture connecting a master with several slaves [byt]



(b) SPI architecture connecting a master with one slave [byt]

Figure 2.15: SPI architectures for multiple (2.15a) and single (2.15b) slave connection [byt]

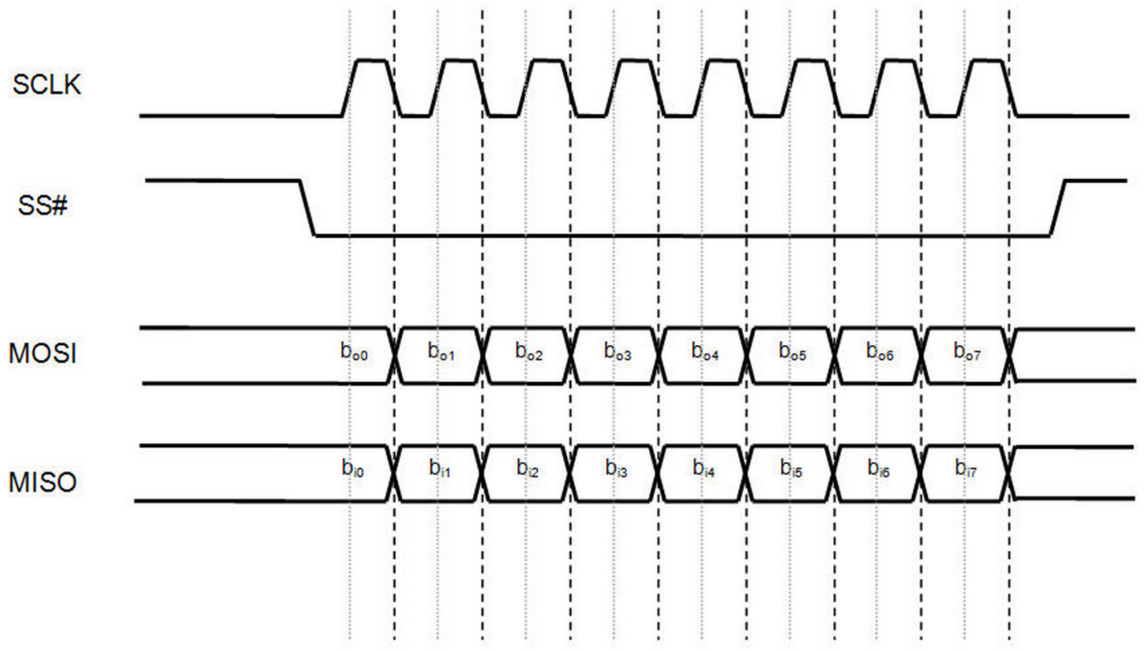

SCLK
SS#
MOSI
MISO
b_o0 b_o1 b_o2 b_o3 b_o4 b_o5 b_o6 b_o7
b_i0 b_i1 b_i2 b_i3 b_i4 b_i5 b_i6 b_i7

Figure 2.16: SPI communication example that shows data toggling and sampling [byt]

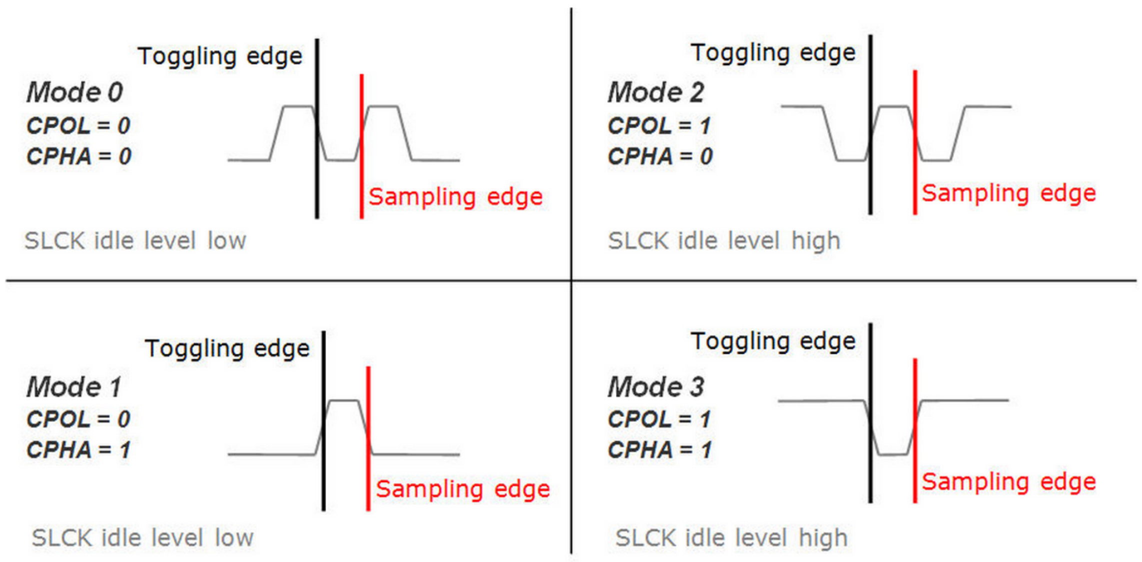


Figure 2.17: The four possible modes used in the SPI specification [byt]

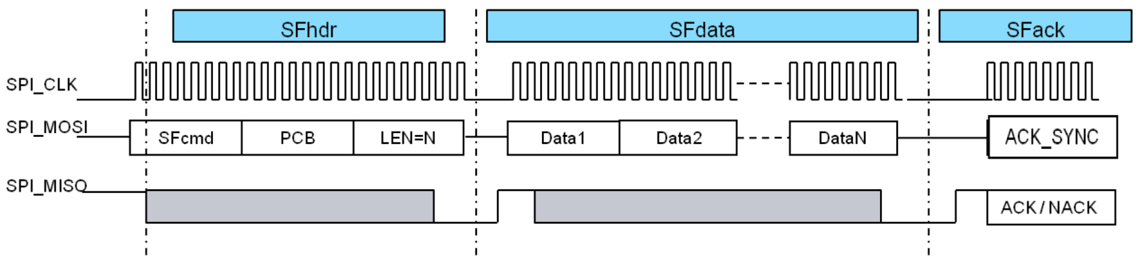


Figure 2.18: The SPI frame format forming a new protocol layer SF [an-11, page 9]

Data Link Layer

As the SPI standard defines no definition of a MAC layer, it is often implemented by the vendor itself. One variant is the SPI Framing Layer (SF) used by NXP [an-11].
This layer is - similarly to the SWP protocol - responsible for keeping master and slave in sync, the error detection regarding block length or clocking and for resynchronization. In Figure 2.18, the frame format can be seen. It consists of three parts [an-11]:

SFhdr:
: The SPI frame header is mandatory. It is three bytes long containing the SF command (Write / Read data, Power Down, etc.), Procotol Control Byte (PCB) (a byte constant used for additional info as whether a Quick Command is used, Wakeup Command or Error Codes) and the length of the data to be transferred in the SF data field.

SFdata:
: An optional SPI frame data field containing the data to be transferred that can be split into several sub-frames. The number of sub-frames is dependent on the number of data bytes present as well as the SPI sub-frame size supported by the slave. The default sub-frame size is 16.

SFack:
: The SPI frame acknowledge - which is also mandatory - is a one byte string appended at the end of each frame used to acknowledge or not acknowledge a command or data transmission. Thus, it is used for synchronization.

An additional high-level data link layer is then used to organize the data bytes that are transferred. This could be the same or a similar one as the Simple High-level Data Link Control (SHDLC) protocol used by SWP described in Subsection 2.2.1. As this is a de-facto standard where each manufacturer can specify the protocol to its applications' needs, a detailed description of this layer does not make any sense. In fact, a similar overhead as for the SWP arises for this additional layer.

Transfer Rate

The transfer rate is dependent on the clock speed the slave supports which is usually a few MHz. The maximum bitrate is up to 12 $Mbit/s$ as stated in [an-11].

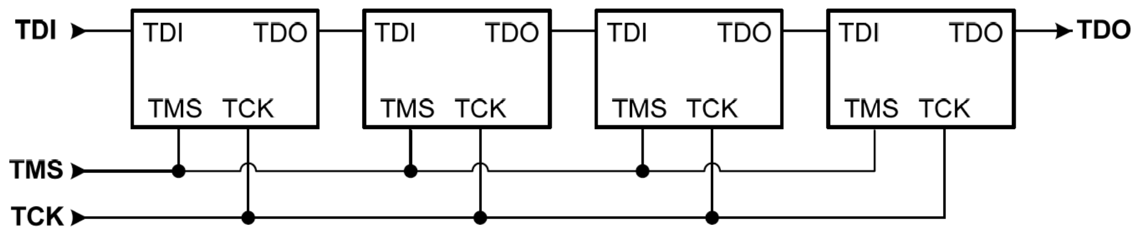


Figure 2.19: JTAG boundary scan architecture used in a daisy chain style connecting on-chip components [A+01, page 18]

However, the marshalling costs have to be considered again, as these numbers only represent the raw bit-rate including the redundant protocol-related overhead. For the data link layer presented in Subsection 2.2.2, these are 4 header *bytes* for a maximum of 255 data *bytes*[8] [an-11].

2.2.3 Joint Test Action Group

Joint Test Action Group (JTAG) is the widely used term for the Institute of Electrical and Electronics Engineers (IEEE) 1149.1 standard [A+01] which describes the *Standard Test Access Port and Boundary-Scan Architecture*. This standard has been devised by a consortium of big electronic companies to test their ICs when they are already assembled and soldered on the so-called *printed circuit board*. Its most famous application is the boundary scan chain which can be seen in Figure 2.19, where multiple HW components are connected in a chain over a serial connection with only one Test Mode Select (TMS) signal being used.
This allows to put some test-data with a desired clock frequency on the JTAG interface and test one IC after the other as the output (Test Data Output (TDO)) is connected to its neighbor's input (Test Data Input (TDI)). [jta]

When using JTAG to connect two HW modules - as supposed for our target platform - the output of the first component is used as input of the second one and vice versa. Thus, the setup reduces to four signal wires TMS, Test Clock Input (TCK), TDI and TDO which is pretty much the same as the setup of SPI (SS, SCLK,MISO and MOSI).
The bitrate is therefore dependent on the higher-layered protocols used on top of this interface. As this timing behavior will be very similar to SPI - both are synchronous serial data connections where the clock speed and the underlying protocol define their throughput - a further description of JTAG is skipped.

More detailed information can be found in [A+01]. There all the possible board-level interconnection alternatives of components are shown and a detailed explanation of the Test Access Port (TAP) controller[9] and the various registers[10] is given.

[8] 3 bytes for SFhdr, 1 byte for SFack and a maximum of 255 bytes for SFdata.
[9] A finite automaton - controlled via the TMS and clocked by the TCK signal - that is responsible for shifting the inputs and outputs through the daisy chain.
[10] I.e. instruction, bypass, boundary-scan and test data registers.

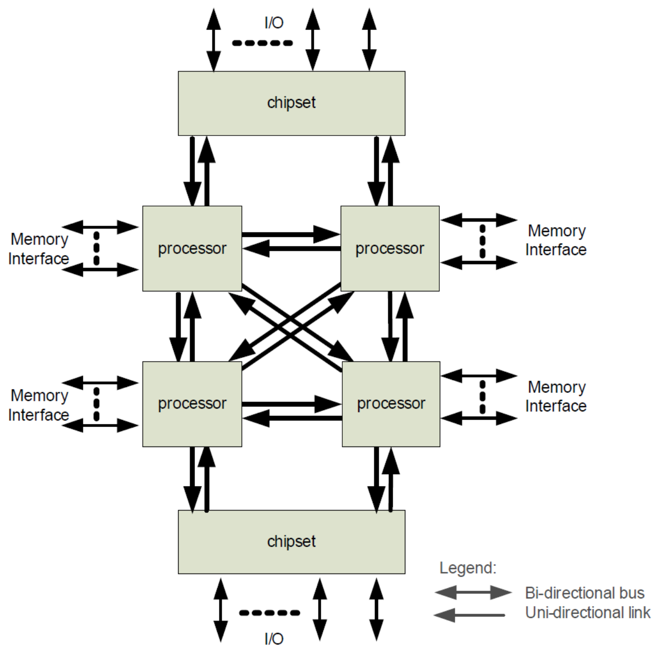


Figure 2.20: Overview of Intel's Quickpath setup showing the direct-link connections between two corresponding units [inta, page 8]

2.2.4 Intel QuickPath

The Intel Quick Path (IQP) proposed in [inta] is the successor of the Front Side Bus (FSB)- which has been used in desktop PCs in the 1990s and 2000s - and is a high-speed point-to-point connection scheme. Due to the evolution of on-chip communication with its higher traffic amount also the classical FSB had pushed its envelope. A shared-bus for all the processors being used at the same time could not cope with the increasing amount of data to be transferred thus leading to a split up. First, the traditional FSB was divided into two buses in the year 2005. Later on the evolution led to the IQP with its dedicated connections between two components as can be seen in Figure 2.20.

Physical Layer

The actual wires used for connecting two components with each other are illustrated in Figure 2.21. One interconnect port consists of four uni-directional links operating simultaneously, forming two link-pairs for each direction. Each link consists of twenty 1-bit lanes as data signals and one forwarded clock. The data signal pairs are Direct Current (DC)-coupled[11] and use a differential signaling[12] scheme. This leads to a total of eighty-four signals to form a single IQP interconnect port.[inta]
Operational data-width is specified as full, half and quarter widths which is specified

[11] Thus, both, the Alternate Current (AC) as well as the DC part of the signal is allowed to pass through the connection. This means the signal sent over the wire is not affected at all.

[12] Two complementary signals are transmitted over two paired wires - the *differential pair*. This results in a very robust transmission technique with respect to external interference such as elegromagnetic noise since information is sent as difference between the wires.

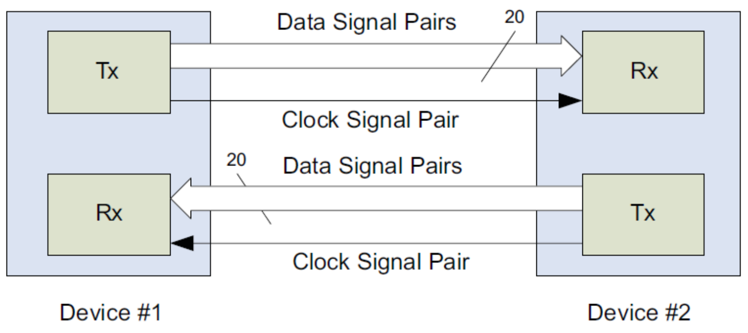


Figure 2.21: Architecture of one Quickpath connection between two components with its various wires [inta, page 10]

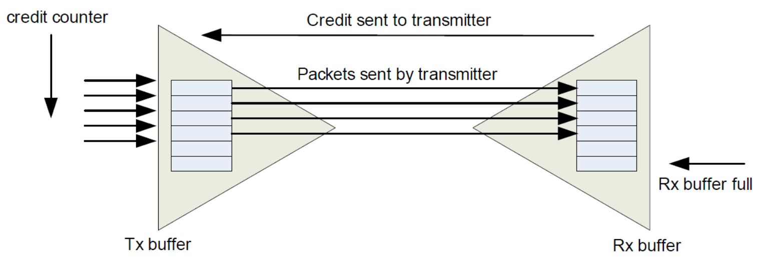


Figure 2.22: Credit / Debit flow control used on IQP's data link layer [inta, page 13]

during intialization. Therefore, all the information being transferred on one clock edge - called a *Phit*, physical unit - could be 20, 10 or 5 bits, respectively.

Data Link Layer

On IQPs' data link layer so-called *Flits* - flow control unit - with a size of 80 bits each, are transferred. Thus, depending on the operational data-width discussed in Subsection 2.2.4, the number of Phits that have to be transmitted to represent a Flit are 4, 8 or 16, respectively.
As the flow-control is one task of the data link layer, IQP uses their credit / debit scheme as depicted in Figure 2.22. During initialization the number of credits to send Flits to the receiver is determined. Whenever a Flit is sent to its receiver, this credit counter is decremented. When it reaches zero, IQP stops transmitting data over that channel. Also, whenever the receiver has read out a buffer and freed it, it returns credits to the transmitter, which in turn is then able to transmit Flits again.
In order to compensate or detect routine errors occuring on the physical layer, a CRC check is performed by the data link layer. The CRC is generated at the Transmission (TX) side and checked at the Reception (RX) side. For this purpose, 8 bits of the 80-bit Flit are reserved.
The protocol-stack of IQP also defines even higher layers, the *Routing Layer*, *Transport Layer* and the *Protocol Layer*. As these layers are not considered for this work's evaluation, for a more detailed description please refer to [inta, pp. 14].

Transfer Rate

The theoretical bandwidth of IQP is 32 GB/s[13]. However, in [inta] they state it with a more realistic value of 25.6 GB/s which is twice the amount of the older FSB. For this calculation only 16 of the 20 data lanes of each link are considered as only these are used for "real" data payload[14]. Additionally, higher layer protocol overhead has to be considered for these numbers.

2.2.5 Summary

In this chapter various distributed-memory based direct-link connections have been presented. Focus has been layed on covering most of the currently present on-chip point-to-point connection solutions. With SWP a very simple, low-area one-wire solution has been presented. Its protocol is very robust against errors and has already a flow-management included. Also the support for contactless communication via NFC is already supported by its protocol.
With JTAG and SPI two similar communication architectures are introduced which offer higher througput but also at higher HW costs since additional pins are needed on the final package. Another similar approach would be the Inter-Integrated Circuit (I^2C) bus which is multi-master, uses two-wires but has a bandwidth being only in the range of $3.4 Mbit/s$ in its high-speed variant. I^2C offers advanced features (multi-master conflict handling) which means it is rather complex and therefore lacks in performance. [byt]
With IQP, a high-speed interconnect has been shown. It is similar to other known alternatives such as PCI-Express. As IQP has better performance characteristics compared to PCIExpress [inta] and it is the successor of the common known FSB and thus in use in current desktop systems, IQP has been presented here.

The properties of all direct-link connection models can thus be summarized as:

+ Highly scalable:
  Due to the direct-connection nature of these communication schemes it is easily possible to scale up with additional components to be connected. Although, this means additional wires are introduced as well of course.

+ Synchronization:
  The synchronization is handled implicitly in the protocols.

- Marshalling costs:
  One drawback is definitely the additional overhead introduced by the underlying protocols - the so-called marshalling costs.

- Medium transfer-rates:
  Except for the IQP, all introduced variants only offer low- or medium-transfer rates.

In Table 2.1 a very high-level classification based on rough estimations is illustrated. This comparison is only among the distributed-memory based architectures presented in

[13] At a clock frequency of 3.2 GHz - double pumped, thus 6.4 GT/s, with 2.5 bytes per transition and two transitions - for both directions - at the same time, this results in $32GB/s$.
[14] Due to CRC- and other protocol-related-overhead.

this section. It should provide the basis for the design decision on which one of the direct-link communication interfaces fits best with respect to the target platform described in Section 1.1.4.
Due to the fact, that a smartcard HW development process is mainly area driven, it is reducing this property which is of highest priority. Each of the communication approaches consists of some wires and some HW-logic implementing their behavior. However, where digital HW shrinks in size very well for each new process technology, the size of the interconnect-wires does not. Every signal that is led out of a μC results in a *pad-area* inside the IC where the *bonding-wires* are attached on. It is *not* possible to arbitrarily shrink these pads due to physical reasons. Therefore, the *Complexity* indicator is defined by the number of wires a communication architecture consists of.
The *Throughput* is the theoretical raw bandwidth which is calculated from the *Clock Frequency* times the bits-per-clock-cycle times the number-of-directions data can be sent simultaneously.
In the *Clock Frequency* column the maximum clock frequency of a communication architecture - as defined in its dedicated specification - is listed.
The *Marshalling Costs* reflect the overhead introduced by the physical as well as the data link layer to guarantee an error-free data transmission. It is this the relation of header-bytes to data-bytes.
Due to the same reasons the JTAG approach has not been elaborated further as mentioned in the corresponding subsection, it is not listed in Table 2.1.

Interface	*Complexity* [$\#wires$]	*Throughput* [$Mbit/s$]	*Clock Frequency* [MHz]	*Marshalling Costs* [$\#header_{bytes}/\#data_{bytes}$]
SWP	1	2	-	10 / 58
SPI	4	24	12	8 / 510
IQP	84	$256 * 10^3$	$3.2 * 10^3$	1 / 4

Table 2.1: Comparison of the distributed-memory based communication interfaces

2.3 Shared-Memory based Models

In Section 2.2 the communication architectures relating to the distributed-memory based approaches have been discussed. According to the classification of Figure 1.3, the other main category describes the shared-memory based architectures.
Shared-memory based approaches are normally bus-based with a global address space allowing both sub-systems to communicate over a dedicated, shared-memory. Again, different variants are shown - each with different levels of HW complexity suiting for different target systems - and summarized at the end of this chapter. A decision on which approach to use for the target platform is done in Chapter 3 while a quantitative comparison between the selected approaches is then performed in Chapter 5.

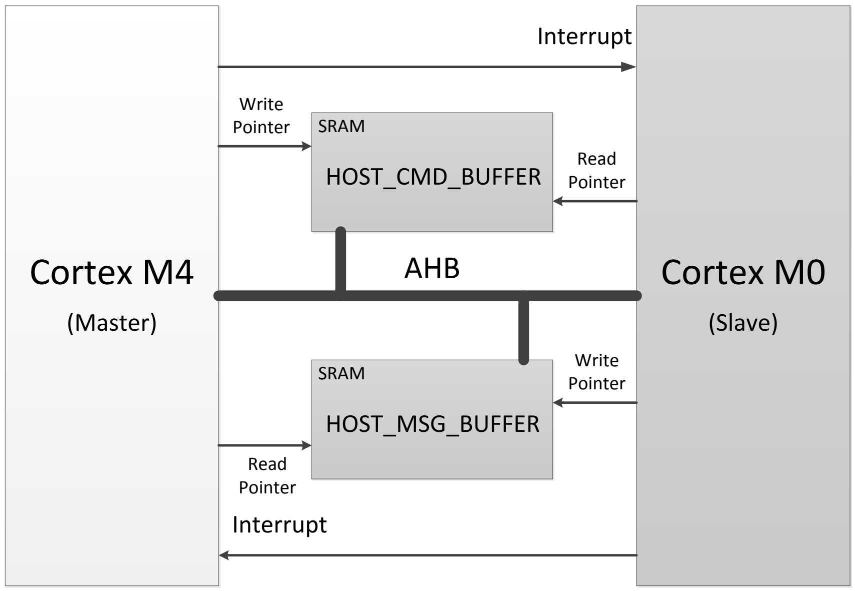


Figure 2.23: NXP Semiconductors' IPC architecture connecting two ARM Cortex CPUs via shared memory attached to a bus [an1, page 10]

2.3.1 NXP Semiconductors Inter Processor Communication

The shared-memory bus-based communication approach that can be seen in Figure 2.23 defined in the application notes [an1], NXP Semiconductors describes a communication architecture connecting two ARM-CPUs. One of them (Cortex M4) being the communication master and the other being the communication slave.
There is a dedicated memory region for the communication in each direction and so-called pointers describing the actual write and read positions inside the memory. These are also used for synchronization, since the read pointer always has to be "before"the write pointer in the sense of a circular counter. Thus, the memory is building some sort of circular buffer, eliminating the needs of shifting data after a read or write.
The interrupt wires can be used to signal incoming data - either after each sent data word or after multiple data words being sent which would reduce the interrupt-related overhead introduced in the receiving CPU.
An alternative version of this approach with very little HW costs would be to just use the interrupts for notifications without any memory. However, this would only allow to notifiy the other CPU which could then perform some event-related work without any additional data being transferred. [an1]

Physical Layer

An IPC block - describing the part of the module being responsible for one direction - consists of:

- A memory region for storing the data.

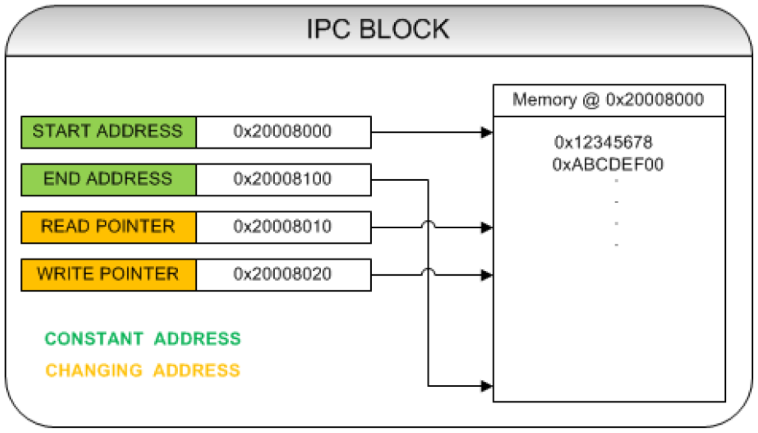


Figure 2.24: One NXP Semiconductors' IPC block composed of a memory buffer and four registers [an1, page 11]

- A register holding the start address of this memory region.
- A register holding the end address of this memory region.
- A register holding the read-pointer, thus the address of the next data to be read by the receiving CPU.
- A register holding the write-pointer, thus the address of the next data to be written by the transmitting CPU.

In Figure 2.24 one can see how the memory area and the four previously stated registers are related to each other. The memory buffer used for one communication direction is only writeable by the transmitting CPU and only readable by the receiving CPU. An additional HW mechanism controls that the write-pointer never gets equal to or greater than the read-pointer. Equal write- and read-pointer indicate an empty buffer. [an1]

Transfer Rates

The bandwidth of this communication scheme is directly dependent on the bus-clock speed used for the shared-bus. The additional delays introduced by the bus-arbitration are negligible in case only these two entities are connected via the bus and only one of them is trying to communicate at the same time. This assumption holds for a lot of applications. So one side puts all its data inside the shared-memory indicating a finished transmission via the interrupt, resulting in the slave to read out the data.
Address-decoding is also negligible due to the pipelining feature of the Advanced High-performance Bus (AHB). At least for the case when multiple words are transmitted at once, which is also true for the considered applications.
Assuming a bus-clock speed of 100 MHz and a bus-width of 32 $bits$, the resulting theoretical raw bitrate is approximately 6.4 $Gbit/s$ as data can be exchanged in both directions simultaneously.

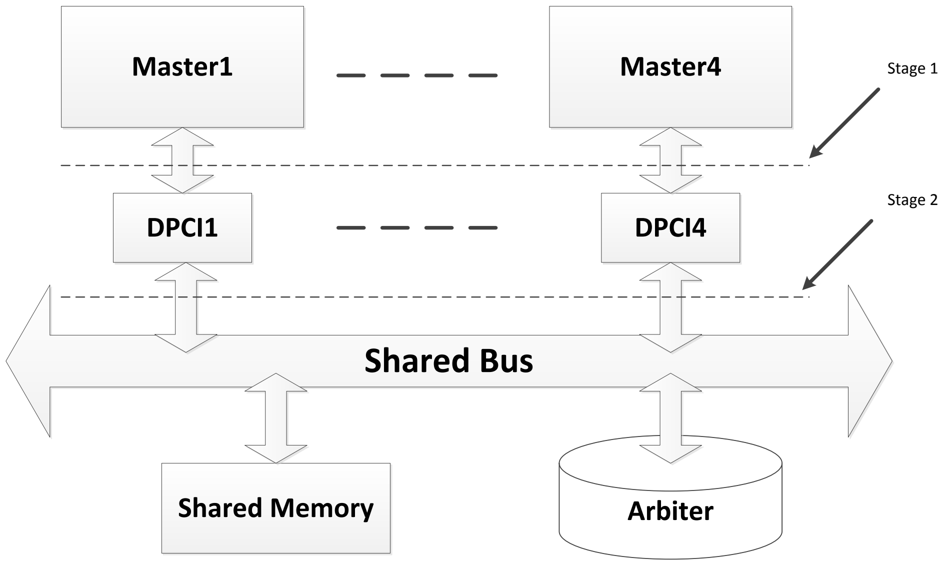


Figure 2.25: Architecture of the DPCI bus-based shared-memory communication approach [dpc]

2.3.2 Data Pre-Fetch Core Interface

An interesting, highly scalable shared-bus concept is presented in [dpc]. A new component called Data Pre-fetch Core Interface (DPCI) is put in between the master communication components and the shared-bus. An overview is given in Figure 2.25:

The architecture of this approach has two advantages: it serves as a buffer eliminating the problems that arise through the individiual clock domains of the master-components and the shared-bus. Additionally, a data pre-fetching unit is part of the DPCI which allows a better bus-utilization by accessing the bus in advance when its free. The data pre-fetch unit uses the fact that related data is often placed in memory next to each other. Therefore, when one data word is requested by the master, the DPCI automatically reloads the next couple of words of the shared-memory location into its own buffer when the bus is not used.

Physical Layer

As depicted in Figure 2.26, the DPCI consists of a write buffer, a read buffer and a config unit. The write and read buffers are used to serve as buffers for the individual clock domains of the single master units as well as by the pre-fetch unit for a further increase of system efficiency as described in Subsection 2.3.2. The config unit is used to allow new Intellectual Properties (IPs) being added in a plug-and-play fashion, just by editing the configs after they have been attached.

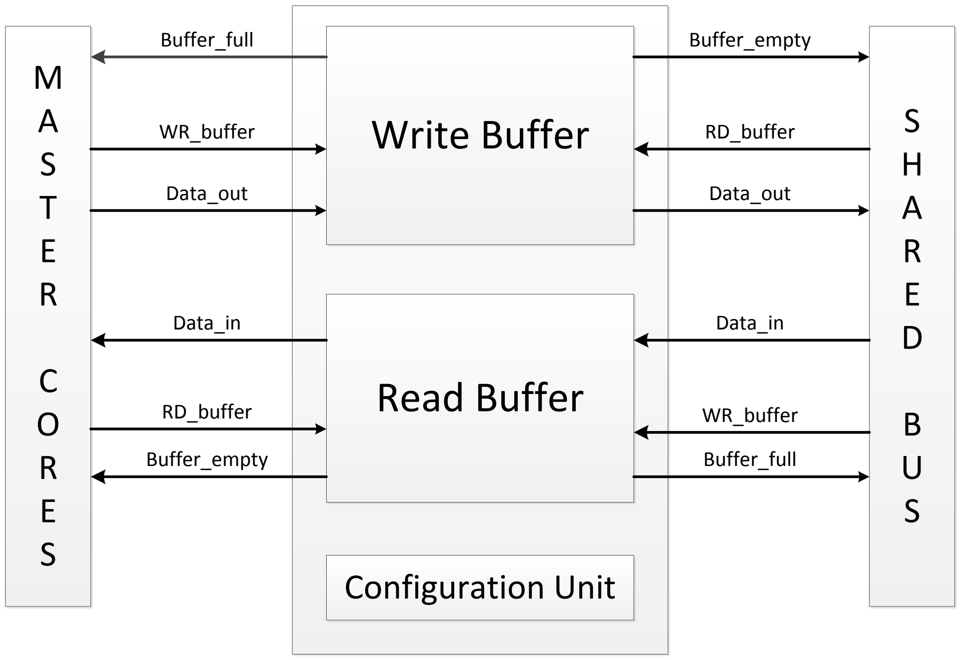


Figure 2.26: The physical setup of the DPCI with its buffers, config unit and signals [dpc]

Transfer Rates

This approach has another factor additional to the bus-clock speed influencing the transfer rates, namely the additonal stage introduced by the DPCI between the master-components and the shared-memory. Thus, everytime data is exchanged between a master- and a slave-component, it has to be copied from the master's memory into the write buffer, then from the write-buffer into the shared-memory and afterwards again from the shared-memory to the read-buffer of the slave-component's DPCI unit and finally from there to the slave's memory. This results in a delay of at least 4 $Cycles$ which is especially relevant for small data-chunks that are exchanged.
The architecture's theoretical raw bandwidth is 3.2 $Gbit/s$ for a typical bus-clock frequency of 100 MHz since data can only be exchanged in one direction simultaneously.
Obviously, this approach is a trade-off solution between scalability versus delay. This means, it allows additional components to be attached in the system on a shared-bus with a better bus-utilization but on the other hand introducing additional delays due to the two-stage scheme.

2.3.3 Hierarchical Bus-based External Communication

A hierarchical-bus-based shared-memory communication approach called *ExtComm* is presented in [Vie14]. As can be seen in Figure 2.27 the two sub-systems are connected via an additional bus. An additional register used for synchronization called *InterCommLock* is also attached to the hierarchical-bus in between the sub-systems.

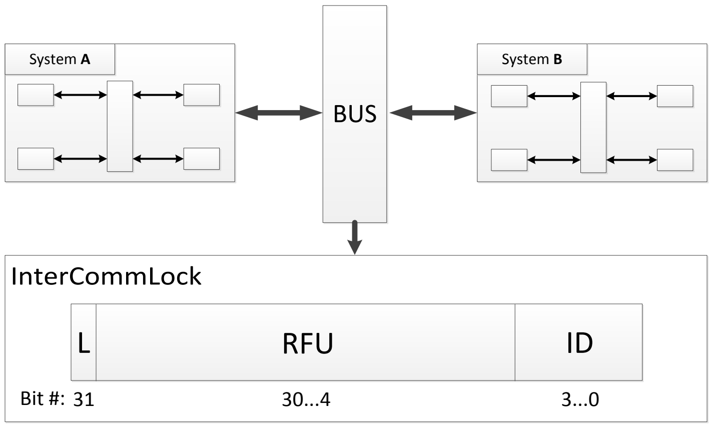


Figure 2.27: Setup of the ExtComm communication approach used in [Vie14, page 16]

Physical Layer

The ExtComm component itself consists of some shared memory, a status register holding the actual status of the component and its interrupt-status and a control register used to control the states of the component.
The ExtComm components are being intitialized by the transmitting systems' CPU to get into their proper states (receiving / transmitting). This is done via the InterCommLock register to prevent both systems to start transmitting at the same time. Then the data is copied into the foreign systems' ExtComm unit.

Transfer Rate

The theoretical raw bandwidth of this approach is 3.2 $Gbit/s$ at a bus-clock frequency of 100 MHz as data can be transacted only in one direction at the same time.
Thus, the transfer rate of this approach is slower than the Inter Processor Communication (IPC) approach of Subsection 2.3.1 due to the additional overhead introduced by the Finite State Machine (FSM)-logic. Also the additional bus leads to a further delay. Both of these effects get more and more negligible as more data is transferred in one transaction. All in all, the delay for the first data-word to be written is at least 10 $bus - cycles$.
This solution allows multiple sub-systems to be easily connected through the hierarchical-bus solution while introducing another bottleneck - the top-level bus where the InterCommLock register is attached to.

2.3.4 Summary

The shared-memory communication architectures presented in this chapter describe actual state-of-the-art concepts either being in use as NXP Semiconductor's IPC variant, or concept-studies as DPCI or the ExtComm approaches. Of course, these architecture

variants raise no claim to completeness, however bus-based shared-memory approaches are all quite similar. Depending on the number of components they communicate with, they are more (DPCI) or less (IPC) complex.
Additional variablity arises with the connection topology as seen in the ExtComm as well as the DPCI solution where a hierarchical-bus scheme is used. The topology-related considerations are also dependent on the number of components being connected to each other. A hierarchical-bus system allows more units to be connected over wider distances but causing additional delays related to the bus-bridging, different clock-speeds etc.

Common aspects of the shared-memory communication scheme can be summarized as follows:

+ No marshalling:
  Due to the global address space and the underlying shared-memory with concrete addresses the data is placed in, no additional protocol is needed (besides the bus-protocol).

+ Efficient data transfer:
  Huge chunks of data can be transferred very efficiently due to the typically wide bus-widths of e.g. 32-bits and more.

- Synchronization mechanism needed:
  Some additional HW effort is needed for the synchronization which is not done implicitly by the protocol anymore (in contrast to distributed- memory).

- Limited scalability:
  Obviously, the bus connection is the bottleneck when the number of so-called communication-master-entities is rising.

In Table 2.2 a very high-level classification of the shared-memory based communication architectures based on rough estimations is illustrated. It summarizes the properties of the shared-memory based interfaces with respect to the design decision on which one to select for implementation in order to find the best solution for to the target platform described in Section 1.1.4 .
Since the most important factor for a smartcard-interface is its needed area, these indicator is abstracted in the *Complexity* column. As described in Subsection 2.2.5, the most influencing factor for the needed area is the number of wires needed by the approach. Due to the fact that these numbers are highly dependent on which bus is used, it is this factor abstracted to the number of layers each communication interface is built of. The amount of layers indicator indirectly reflects the number of wires and thus, the HW-area-effort of the shared-memory based communication architectures very well.
The theoretical raw bandwidth of each approach is listed in the *Throughput* column, which is calculated through the *Clock Frequency* times the bus-width times the number of directions, data can be transferred simultaneously.
The *Clock Frequency* column contains typical bus-clock frequencies for smartcards.
A higher number of delay-cylces - for the first word written into the shared-memory - of the DPCI and the *ExtComm* solution are due to the additional layers and the FSM-related overhead, respectively. These delays are more and more negligible as the amount of data

sent at once for each transaction is higher and higher. However, applications where often short commands or responses are exchanged would suffer from these higher delays.
Finally, the *Scalability* refers to the degree whether each communication architecture is able to connect more than two components with each other, thus to cope with a higher amount of data.

Interface	*Complexity* [*#Layers*]	*Throughput* [*Mbit/s*]	*Clock Frequency* [*MHz*]	*Delay* [*#Cycles*]	*Scalability*
IPC	1	$6.4 * 10^3$	100	1	very bad
DPCI	3	$3.2 * 10^3$	100	4	good
ExtComm	2	$3.2 * 10^3$	100	10	bad

Table 2.2: Comparison of the shared-memory based communication interfaces

Chapter 3

Design

As already mentioned in Subsection 1.1.4, the focus of this work lies on the evaluation of the communication among the sub-systems while performing different applications.
In Figure 1.5 the basic parts of the evaluated system are shown. One sub-system could be for example an NFC-controller, with all the logic needed for the external communication of the card. This includes the analog front end for the RF-communication and a FSM being able to interpret and respond to incoming commands. The other sub-system could be a so-called Secure Element, holding very sensitive data such as the private key - the key needed for all the encryption and decryption performed by the smartcard.
Although, nowadays these functionality can be integrated inside a single microchip, it is very likely that they will be separated into multiple microchips, each for a very specific purpose. These microchips are then integrated in a single package. This way, the single microchips can be developed and produced independently which introduces great potentials of cost savings and re-use.
Therefore, the overall system used to make the measurements and gain the results, consists of two main parts:

- The application models, representing the underlying applications with their differing commands and time-limits - given by the communication protocol or by the implemented standard.

- The hardware models, representing the communication architecture with their underlying protocols.

Starting point of Chapter 3 are the design considerations in Section 3.1 - split into the simulation setup overview, the benchmark- as well as the modules-part.
In Section 3.2 the single applications with its different properties regarding their communication traffic behavior are presented. This implies the transaction flows describing the commands and responses being exchanged between the sub-systems, as well as the typical maximum delays being allowed for a specific application.
The third part of this chapter describes the architecture of the evaluated communication approaches. Also the way how the underlying protocols are abstracted for the model to remain time-accurate while allowing a simulation speedup is presented in Section 3.3.

3.1 Design Considerations

At the very beginning the overall application requirements with their main properties and challenges is presented in Subsection 3.1.1.
The simulation setup describes how the measurement results are obtained. In the first Subsection 3.1.2 the overall setup is illustrated with its synergy of the SW- and HW-part. How the benchmark is structured in order to model the various embedded applications in a comparable way is shown in Subsection 3.1.3.
Whereas, the simulation interface for the HW modules to log important points in time of the simulation is explained in Subsection 3.1.4.

3.1.1 Application Requirements

As the resulting system consists of SW as well as HW, for the evaluations each application's requirements and needs have to be specified first. The evaluated applications that are at the center of attention can be classified as follows:

Banking:

A typical state-of-the-art banking application with offline-terminal authentication - conforming to the Europay international Mastercard Visa (EMV) standard *EMVCo* [emv] - is evaluated.
The main property of this application is its very strict time-limit of about half a second for the whole transaction since this directly affects the customer comfortability.
As the communication profile consists of several command-response pairs of only a few hundreds of bytes coupled with some processing in between, it is of high interest whether a communication architecture with a higher throughput justifies its bigger area requirements.

Smart Metering:

Due to the lack of an upcoming standard for smart metering applications by the time this work has been written, this application reflects a possible future solution with a hybrid cryptography scheme that is very likely to be used in the near future for such applications. Due to laws governing data protection and data security for applications that process user-privacy related data, an encrypted exchange of this data is inevitable.
The communication needs of this application are thus defined through the encryption overhead as well as the structure of the measurement-data.
As the measurement data is being exchanged between a host and the smart meter according to some pre-defined time-schedule without any user-interaction or strict response-time requirements[1] and a communication volume of about a few kilobytes, the evaluations will show which communication architecture is more appropriate.

Secure software update:

As the smartcard applications' complexity is permanently rising, also additional needs come up for the maintainability and exntensibility of these systems. By now, the functionality of the smartcards has been programmed into their ROM at the

[1] Two consecutive measurement-data requests are seperated by e.g. 24 hours.

beginning of their lifetime and has not been changed until the end of it.
With an update possibility additional use-cases arise, as for example a customer who is now able to perform a security-related update on his own. Probably this will be done through an RF-reader with the customer holding the smartcard in its field during the whole update process. Thus, this procedure has to be finished within a reasonable short period of time to not affect the customer experience in a bad way. However, since today's smartcard OSs can reach memory demands of a few hundreds of kilobytes without any problem, this is quite a challenging task for the underlying communication architecture. Together with the application's demand of secure and authenticated exchange of the update-data, finishing the overall update-process within a reasonable amount of time is challenging too.

3.1.2 Simulation Setup

Overview

In order to combine the application's SW and HW part, a well defined simulation setup has to be designed. These parts are depicted in Figure 3.1 at the left- and the right-hand-side, respectively. The benchmark (SW) is cross-compiled to the CPUs' Instruction Set (IS) with the corresponding Integrated Development Environment (IDE). The resulting binary file is then loaded into ROM from where it is loaded by the CPU which then executes the application-specific instructions according to the benchmark source code. Data exchange between two sub-systems is achieved through the *COMM. ARCH* HW-component. This component could be implemented as any one of the communication architectures presented in Sections 2.2 and 2.3, respectively. Further, a configuration file is used to configure all the single HW-components - also independently for each sub-system.
The logging of the simulation is done within a seperate component that is attached to the system-bus in order to be accessible by the SW to log important points of time such as before the transaction of the command or after the reception of the response, etc.

Hardware Components

Due to the properties of embedded systems in general and smartcards in particular as already mentioned in Section 1.1, the selection of a low-power CPU is obvious. Thus, an ARM Cortex-M-series microprocessor is chosen. The CPU model is a LISA-based ARM-Fastmodel with TLM2.0 compliant interfaces that can be connected to further HW-components such as a bus for example.
For the implementation of the various HW components such as the bus, RAM, ROM as well as the communication interfaces, the SystemC 2.2 [$\mathrm{I}^{+}06$] extension TLM2.0 [Ayn09] is chosen.
Since SystemC 2.2. is a commonly used version of SystemC - also within *NXP Semiconductors* - and the new features do not affect this work's implementation as described in Subsection 2.1.3, it is this the version that has been used for the HW-model implementation.

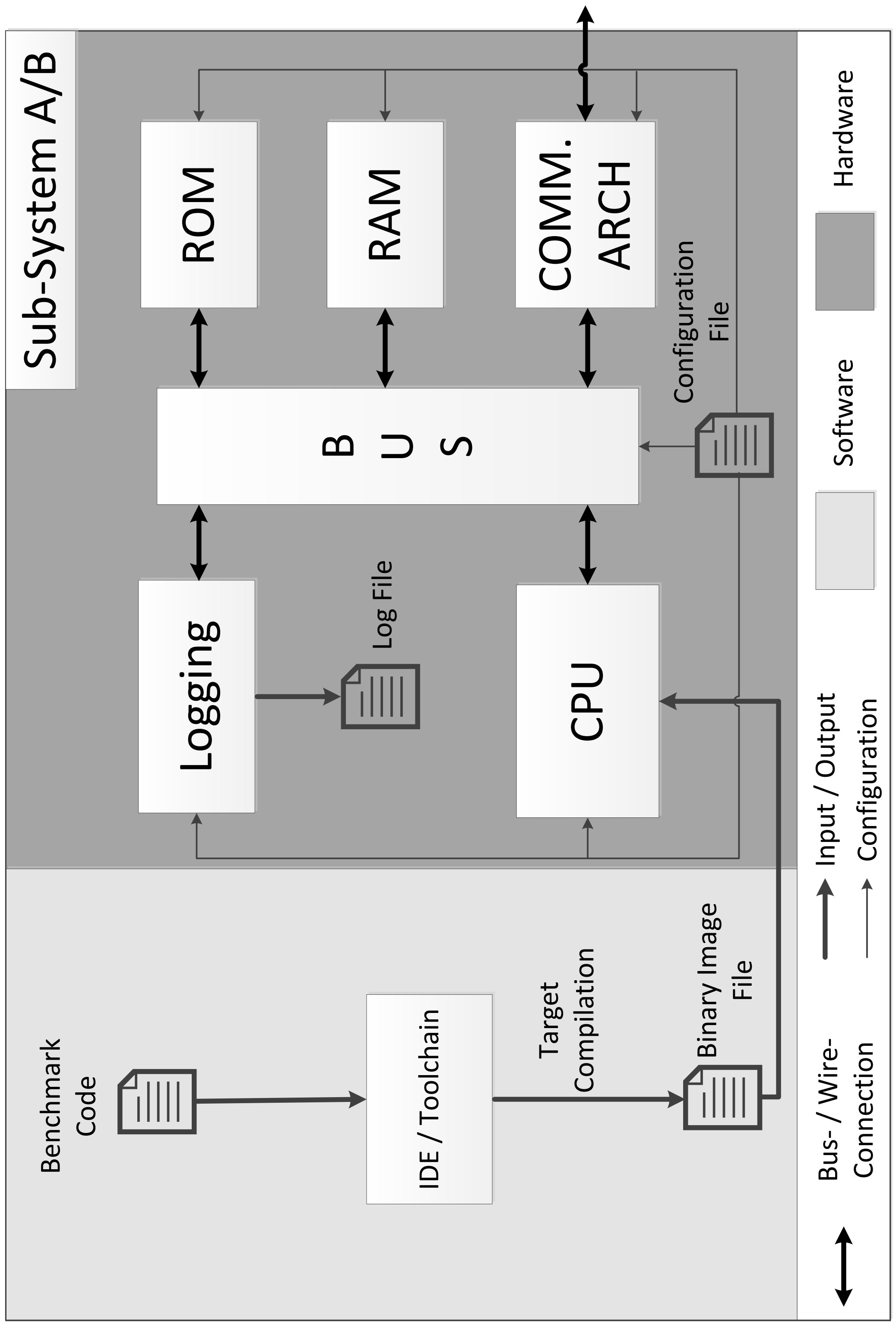


Figure 3.1: Overview of the simulation's setup design for one sub-system

Integrated Development Environment

As development kit the ARM-Microcontroller Development Kit (MDK) - consisting of the *Keil $\mu Vision$* IDE with an ARM C/C++ compiler - is chosen. Although a variety of development toolchains for the ARM Cortex-M family exists, ARM's in-house development kit is widely recognized and of course offers full support for all of their processor-families.

3.1.3 Benchmark Setup

The embedded applications - presented in detail in Section 3.2 - have to be executed on the HW system in a comparable way in order to allow a meaningful interpretation of the results. Thus, the structure of the benchmarks as well as their integration into the overall system is shown in this Subsection.
As already described in Subsection 2.1.5 it is the evaluation of the communication performance which is mainly of interest for this work's evaluations. However, also the time consumption of the processing part of the overall communication duration is considered. It is this modelled as a delay according to the specific calculations that take place between two concecutive transactions.

Benchmark Generalization

Depending on the actual application a benchmark should simulate, different numbers of messages and number of bytes per message are exchanged between the two sub-systems. However, all applications share the same pattern: sub-system A sends N messages with different lenghts to sub-system B and vice versa.
Additionally, depending on the calculations being performed, different delay-times have to be considered as these may influence specific communication interfaces' behavior[2].
Thus, from a communication evaluation point of view it is legitimate to abstract the application's benchmark to a list of messages with associated delays for each sub-system varying for each application.

Benchmark Layering

In order to be able to use the same benchmark implementation for all underlying communication interfaces, an abstraction layering scheme has been introduced. The layers are depicted in Figure 3.2.

The advantages of this scheme are the higher accuracy[3], the lower implementation and adaption effort[4] and the smaller error probability [5].
The *Application Layer* represents the application code itself such as a banking application for example. These programs use the Application Programming Interface (API) of the *Hardware Abstraction Layer (HAL)* to perform specific actions such as the transaction of

[2] For the SWP approach for example, a delay introduced by some cryptographic calculations could lead to an *RR-frame* being sent instead of an *I-frame* due to the response-timeout being reached.

[3] No discrepancies can occur as the same implementation is used for all communication architectures.

[4] One application only implemented once.

[5] Due to some initializations for example, that have been performed in one architecture's benchmark implementation but have been forgotten in the other one.

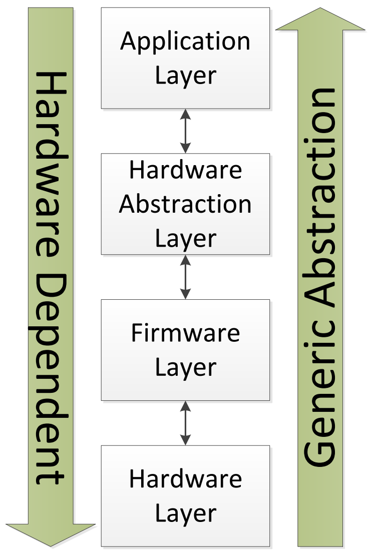


Figure 3.2: Benchmark layered into several layers of abstraction to enable easy switch of underlying communication interface

data to the other sub-system via its internal communication interface. Since the application does not need to know which explicit interface is in use to connect the two sub-systems - as the actual implementation is abstracted through the hardware abstraction layer- the underlying interface can be exchanged very easily.

3.1.4 Modules Setup

To be able to obtain timing-values at very specific moments in simulation time in order to perform a detailed evaluation of an applications' communication behavior, a simulation interface is needed which allows the benchmark executables to initiate a timestamp log.

Simulation API

An additional component is added to the sub-system called Simulation API (*SIMAPI*). This can be addressed from within the benchmark application during simulation to store timestamps in a log-file or print debug messages. As these actions are not considered to influence the timing behavior of a benchmark, accesses of the SIMAPI component are performed without any timing delay.

3.2 Software Application Models

The application models, the programs or benchmarks that represent concrete applications being executed on a smartcard, have been selected in a way that they represent many

different traffic behaviors. Depending on the use case and the specific setup of the two sub-systems, different number and / or lengths of commands and responses between these sub-systems are exchanged. Also the resulting delays are different as this could be only a couple of cycles for e.g. a NVM-write in an update procedure and also a lot more cycles for e.g. a cryptographic calculation as it is used in a banking application to verify a signature for example.
In the following Subsections 3.2.1, 3.2.2 and 3.2.3 the transaction flows including the sequence, the number of bytes and the dedicated delays of the commands and responses being exchanged between the sub-systems, are presented.

3.2.1 Banking

Nowadays, already a big part of the payment is card-based [WR03], thus performed via credit- or debit-cards. These applications are a typical use-case of modern microprocessor-based smartcards.
There are two possible interfaces the card can use for a communication with its environment via the so-called reader as already explained in Subsection 1.1.2: contact-based and contact-less communication. As - by the time this work has been written - the trend of credit-card payments goes towards contact-less payment, the following considerations are all regarding the contact-less protocol defined by the EMV standard [emv].
In order to obtain meaningful results, the benchmark design should be as similar to already used payment applications as possible. There are different specifications available by the time this work has been written but a promising approach is to use the protocol that is used by state-of-the-art banking applications.
The three biggest credit card companies *Europay*, *Mastercard* and *Visa* developed a specification for contact-less payment called *EMVCo* [emv]. Each of these companies has its own specification and own applets which are loaded onto the smartcards. All of them are very similar and mainly differ in the exact sequence of the transactions being performed between the smartcard and the reader.
However, these specifications are not publically available due to security reasons. Therefore, this work evaluates an abstracted, newly designed, typical contact-less banking application. Thus, the benchmark is no real banking applet but a resembled model corresponding to the EMV standard. As the adapted application is modeled to fit the original communication behavior, the resulting simulation values remain valid.

Transaction Flow

In Figure 3.3 the transaction flow of a typical offline-authentication banking application is illustrated. It shows the communication profile between the card-terminal where the card is connected with and the smartcard itself.

The most important steps can be summarized as:

1. SELECT APPLICATION(AID):
   The specific application is selected according to its Application IDentification (AID).
2. FILE CONTROL INFORMATION:
   The card responds the File Control Information (FCI) including the Processing Op-

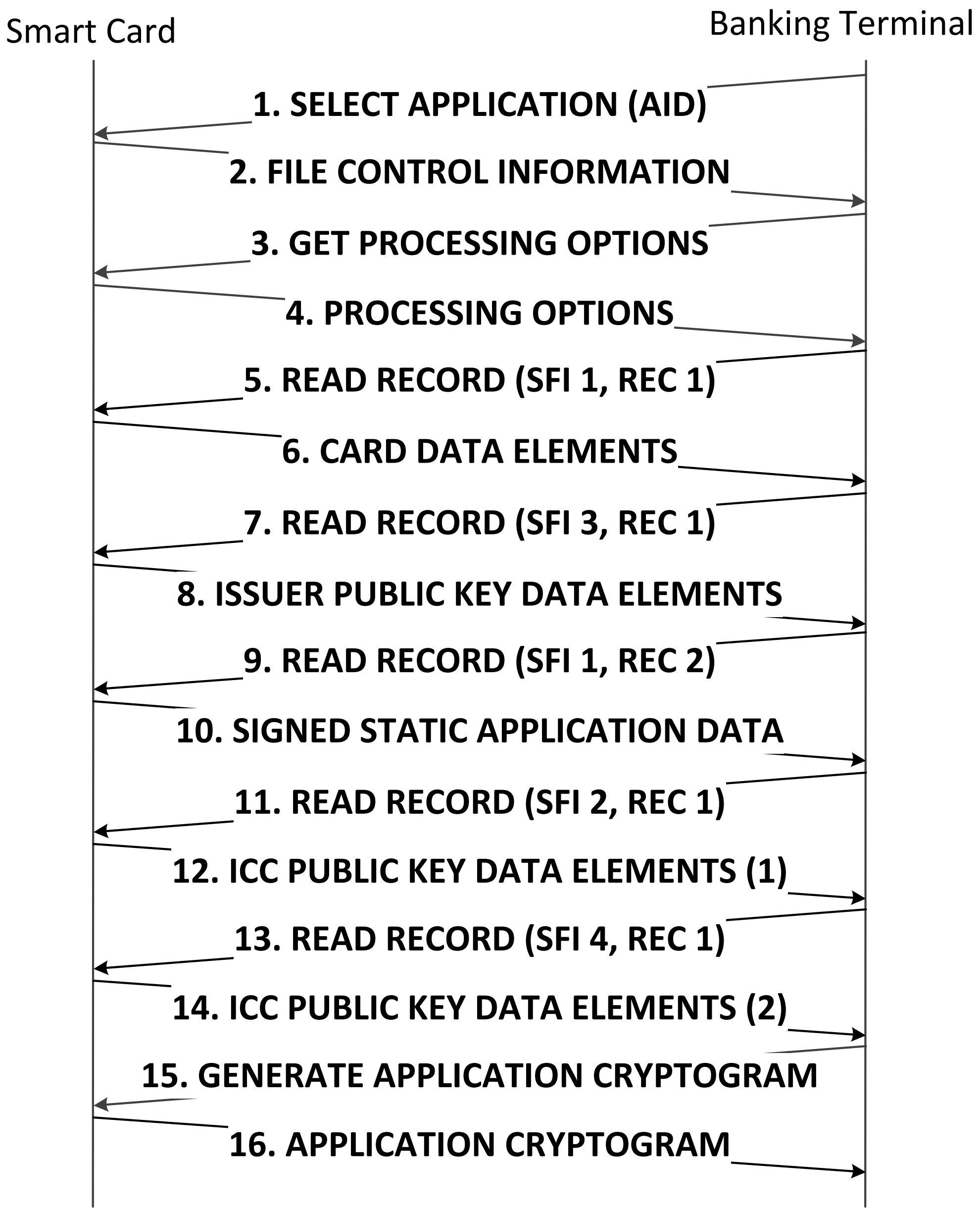

Smart Card
Banking Terminal
1. SELECT APPLICATION (AID)
2. FILE CONTROL INFORMATION
3. GET PROCESSING OPTIONS
4. PROCESSING OPTIONS
5. READ RECORD (SFI 1, REC 1)
6. CARD DATA ELEMENTS
7. READ RECORD (SFI 3, REC 1)
8. ISSUER PUBLIC KEY DATA ELEMENTS
9. READ RECORD (SFI 1, REC 2)
10. SIGNED STATIC APPLICATION DATA
11. READ RECORD (SFI 2, REC 1)
12. ICC PUBLIC KEY DATA ELEMENTS (1)
13. READ RECORD (SFI 4, REC 1)
14. ICC PUBLIC KEY DATA ELEMENTS (2)
15. GENERATE APPLICATION CRYPTOGRAM
16. APPLICATION CRYPTOGRAM

Figure 3.3: Transaction flow of a typical banking application

tions Data Object List (PDOL) which contains a list of tags and lengths of data elements - being located in the terminal - the card needs to process the upcoming Get Processing Options (GET) command.

3. GET PROCESSING OPTIONS:
   With the GET command, the terminal requests the Application Interchange Profile (AIP) as well as the Application File Locator (AFL).
4. PROCESSING OPTIONS:
   The card responds with the processing options supported by the smartcard.
5. READ RECORD(SFI1, REC1):
   The terminal requests the record 1 of the file with Short File Identifier (SFI) 1.
6. CARD DATA ELEMENTS:
   Generic card application data elements such as the application expiry date is sent to the terminal.
7. READ RECORD(SFI 3, REC 1):
   The terminal requests the record 1 of the file with SFI 3.
8. ISSUER PUBLIC KEY DATA ELEMENTS:
   Card data elements to recover the Issuer Public Key are exchanged for the offline data authentication.
9. READ RECORD(SFI 1, REC 2):
   The terminal requests the record 2 of the file with SFI 1.
10. SIGNED STATIC APPLICATION DATA:
    For the static data authentication the Static Application Data is exchanged.
11. READ RECORD(SFI 2, REC 1):
    The terminal requests the record 2 of the file with SFI 1.
12. ICC PUBLIC KEY DATA ELEMENTS(1):
    Part one of the Integrated Circuit Card (ICC) Public Key is transmitted.
13. READ RECORD(SFI 4, REC 1):
    The terminal requests the record 1 of the file with SFI 4.
14. ICC PUBLIC KEY DATA ELEMENTS(2):
    Part two of the ICC Public Key is transmitted.
15. GENERATE APPLICATION CRYPTOGRAM:
    After a specific terminal risk management[6] the terminal decides to either accept it offline, complete it online or decline the transaction offline. For the very first case the generate Application Cryptogram (AC) command is sent by the terminal.
16. APPLICATION CRYPTOGRAM:
    The card performs its own risk management and returns a calculated Application Cryptogram.

[6] Application expiry date checking, terminal floor limit checking, etc.

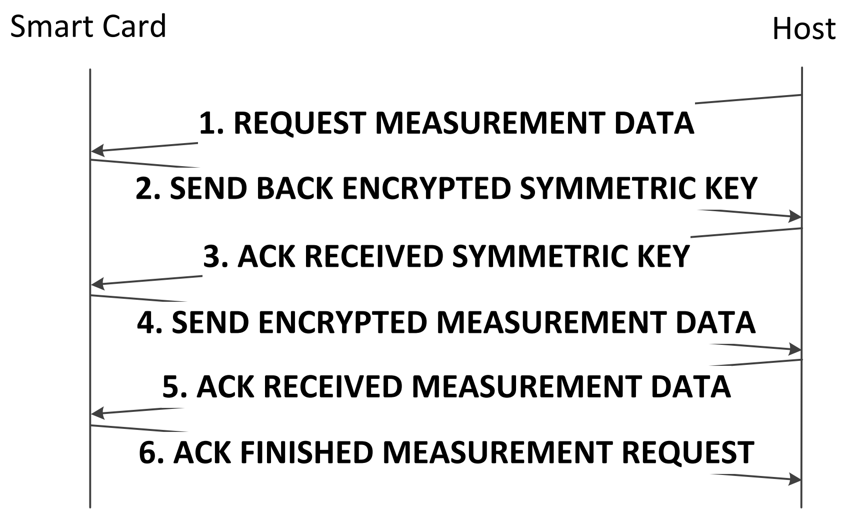

Smart Card
Host
1. REQUEST MEASUREMENT DATA
2. SEND BACK ENCRYPTED SYMMETRIC KEY
3. ACK RECEIVED SYMMETRIC KEY
4. SEND ENCRYPTED MEASUREMENT DATA
5. ACK RECEIVED MEASUREMENT DATA
6. ACK FINISHED MEASUREMENT REQUEST

Figure 3.4: Possible transaction flow of a smart metering application

3.2.2 Smart Metering

At the time this work has been written, no official standards or standardized specifications for smart metering were present. Consequently, the benchmark is designed generically. The basic components of such a system, regardless whether it would be a heating-, power- or water-measurement system, are very similar. In this work, a closer look on a possible power-measurement applications' setup is taken. An upcoming standard would probably only differ in the exact number of commands and / or bytes that are transmitted.

The sequence in such a smart metering system can be reduced to the following steps:

- Once every 10 minutes, the smart meter is performing a measurement thus, read out the current value of the measurement tool and store it in a safe and secure manner.
- Once a day, the Host requests the measured data.

Transaction Flow

In Figure 3.4 the transaction flow of a possible smart metering application between the Host - that is requesting the measurement data - and the smartcard - which has access to the measurement data and is responsible for their secure transmission - is illustrated.

A hybrid cryptography scheme similar to Pretty Good Privacy (PGP) is used to transmit the measurement-data. Thus, the symmetric key is encrypted with an asymmetric cryptography algorithm (e.g. Rivest Shamir Adleman (RSA)) and transmitted to the Host. After the latter has decrypted the symmetric key with its own private key, it is used to secure the exchange of the commands and measurement data with a symmetric cryptography scheme:

1. REQUEST MEASUREMENT DATA:
   The Host sends a measurement-data request which is encrypted with the Hosts private key.
2. SEND BACK ENCRYPTED SYMMETRIC KEY:
   The smart meter verifies the incoming request with the stored public key of the Host.Afterwards, the smart meter sends the encrypted symmetric key to the Host.
3. ACK RECEIVED SYMMETRIC KEY:
   The reception of the symmetric key is directly acknowledged by the Host.
4. SEND ENCRYPTED MEASUREMENT DATA:
   The smart meter starts sending the already encrypted measurement data to the Host.
5. ACK RECEIVED MEASUREMENT DATA:
   The Host acknowledges the reception of the measurement data.
6. ACK FINISHED MEASUREMENT REQUEST:
   Finally, with the last response sent, the smart meter acknowledges the end of the measurement request.

3.2.3 Secure Software Update

Another potential smartcard application is the secure software update of all kinds of SW being stored on the card. This could be either the update of an ordinary application or parts of the OS or even the firmware.
In fact, all of these use-cases only differ in the amount of data that is transmitted. For a typical application this will be only a few hundreds of bytes but when a whole OS-update has to be performed, this could become very time-consuming. Especially, when such a system update is performed via an RF interface such as NFC by an end-user, the resulting time consumption is of very much interest to prevent a negative feedback from the customer.
As these timing values are highly dependent on the underlying communication architecture used to connect the communication controller (sub-system A) with the secure-entity saving the data (sub-system B), this benchmark can be seen as a kind of performance benchmark.

A typical secure software update scheme could be described as follows:

- Authenticate the Host system which is trying to initiate an update of a SW stored in sub-system B's secure memory.
- Retrieve the large amount of update data and store it in the NVM.

Transaction Flow

In Figure 3.5 the transaction flow of a possible secure update application is illustrated. Since the SW that is being updated is security-relevant data, the exchange of the update-data has to be performed in a secure and authenticated way. This is needed to guarantee

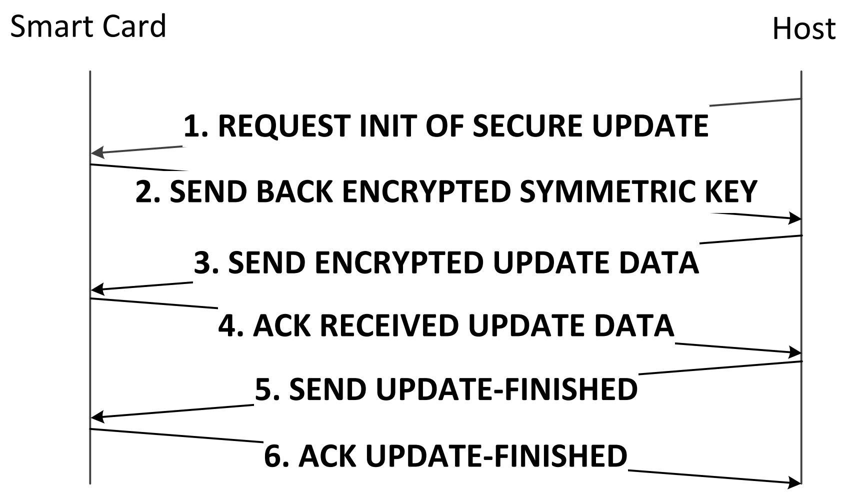

Smart Card
Host
1. REQUEST INIT OF SECURE UPDATE
2. SEND BACK ENCRYPTED SYMMETRIC KEY
3. SEND ENCRYPTED UPDATE DATA
4. ACK RECEIVED UPDATE DATA
5. SEND UPDATE-FINISHED
6. ACK UPDATE-FINISHED

Figure 3.5: Possible transaction flow of a secure update application

an authorized entity is initiating the update and to prevent the data from being altered by an unauthorized entity during transmission.

Due to the typically large amount of update data, again a hybrid cryptography scheme is used. Therefore, the authentication, verification as well as the symmetric-key exchange is the same as in Subsection 3.2.2. The communication sequence is as follows:

1. REQUEST INIT OF SECURE UPDATE:
   The incoming request is signed by the Host with its private key and verified by the smartcard with the Host's public key.

2. SEND BACK ENCRYPTED SYMMETRIC KEY:
   Then, the already generated symmetric-key is signed with the smartcard's private key, encrypted with the Hosts' public key and send back to the Host.

3. SEND ENCRYPTED UPDATE DATA:
   Having the symmetric-key, the Host starts sending the encrypted update data to the smartcard.

4. ACK RECEIVED UPDATE DATA:
   After a successful reception of the update data, the smartcard acknowledges the completely received update data.

5. SEND UPDATE-FINISHED:
   The Host sends the update-finished command to the smartcard.

6. ACK UPDATE-FINISHED:
   Finally, the update-finished command sent from the Host is acknowlegded by the smartcard.

3.3 Hardware Communication Architecture Models

The structure of the applications that are executed on the smartcard has been described in Section 3.2. This Section 3.3 however, describes the communication architecture which connects the sub-systems A and B with each other. As already mentioned at the beginning of this chapter, this is not the communication interface of the card with its reader - which can be either contact-based or contact-less - but the connection inside the package between the two microchips containing sub-system A and sub-system B.
The following subsections contain a detailed description of the structure of the HW components, its underlying protocols and their architecture related impacts on the timing behavior of the models.
In Subsection 3.3.1 the SWP approach is presented and in Subsection 3.3.2 the NXP Semiconductors IPC one.

3.3.1 Single Wire Protocol

With the description of the SWP standard in Subsection 2.2.1, the reasons for choosing this approach as the reference implementation are obvious. SWP is often already integrated in today's smartcard processors and its protocol has been developed to be compliant with the main contact-less protocols used for NFC as well as Radio Frequency IDentification (RFID).
According to Table 2.1 where all distributed-memory based approaches are summarized, it is this approach with the best complexity-to-throughput trade-off. As also stated in Subsection 2.2.5, for a smartcard communication architecture the most important property is its needed area besides the throughput. This is due to the very limited space on a smartcard' IC and the fact, that most smartcard applications have a very limited memory footprint and also communication traffic.

This communication model is a reference model and thus, has to be strictly implemented according to the protocol specification [ETS08].
The design of the SystemC model of the SWP protocol is a little bit different related to its underlying specification. Numerous abstractions (e.g. no framing / CRC needed when using TLM sockets), generalizations (e.g. no *CLT* frame type functionality needed for the evaluation of the on-chip communication) as well as assumptions (e.g.no *ACT* functionality; activation is assumed to be in parallel to external interface activation and thus has no influence on timing behavior) have to be made due to the high-level nature of the SystemC TLM modeling language.
All changes will be listed in the following section including their impact on the SystemC model in terms of timing (delay values), the only important coefficient regarding the communication model evaluation.

Architecture

Figure 3.6 shows the architecture design of the SWP approach. Basically, this consists of three main functional units:

Interrupt Control:

This block is responsible for the surveillance of the module's interrupt status. Thus, the actual status and occured events are compared with the interrupt configuration and if the proper conditions are met, an interrupt is triggered. An example would be the reception of an erroneous frame (e.g. two SOFs occur after another without an EOF in between. Then the proper flag in the interrupt control register is checked and if it is set, the interrupt signal Interrupt ReQuest (IRQ) is set high.

SWP Kernel:

The kernel is responsible for the protocol-specific handling of the data, that is, the MAC-layer part (framing, CRC-calculation) as well as the LLC-layer part (flow control, error management). This block is directly connected to the other SWP-component via a single wire which is used for the actual data transaction. Also the serial-to-parallel conversion and vice versa from the typically 32-bit width registers to the serial wire is performed in this section.

TX/RX Buffer Management:

This unit manages the whole transmitter- and receiver-buffer related work, i.e. updating the buffer pointers to ensure always the correct buffer slot is used for transmission as well as reception of data. Also the correct setting of the associated register-flags such as *data-ready*, which has to be set after some incoming data has been stored in an RX-register and cleared after a transmission from a TX register, is done in this part of the module.

Register Interface:

Represents all the registers of the SWP-module which allows to control and read its actual status and get access to the RX and TX buffers via the system-bus. Also the whole configuration of the SWP module is done via this registers. This includes the filling of the TX registers, the access to the RX buffer data, the behavior in case of erroneous received frames, the interrupt behavior, etc.

The full system with the SWP integration is shown in Figure 3.7. It shows the single wire connection between the two systems via the *S1 / S2* wire. The register access is accomplished via the system-bus of sub-system A and sub-system B, respectively.

Abstractions

- The wire S1 / S2 is abstracted as two TLM sockets - one for each direction.
- Not all registers are considered, only the ones being important for the core functionality as the TX and RX buffers itself, the buffer-control, -status and -length registers as well as the general and interrupt-related control- and status-registers.
- Considered as a timing delay but is not functionally implemented:
  - Bitstuffing - for each additional bit a time delay of *bitduration* is introduced.
  - CRC calculation - not necessary, data is transmitted in an error-free TLM socket.

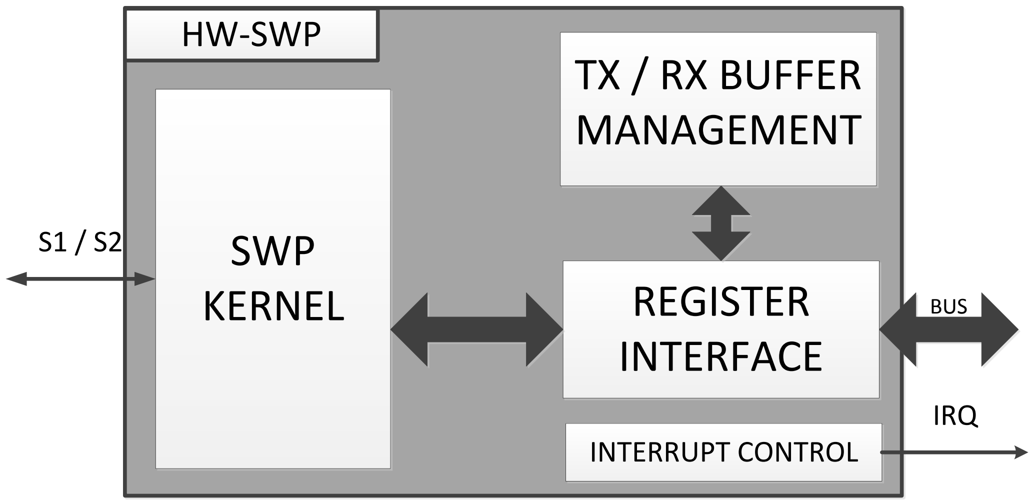


Figure 3.6: Architecture of the SWP module with its functional units

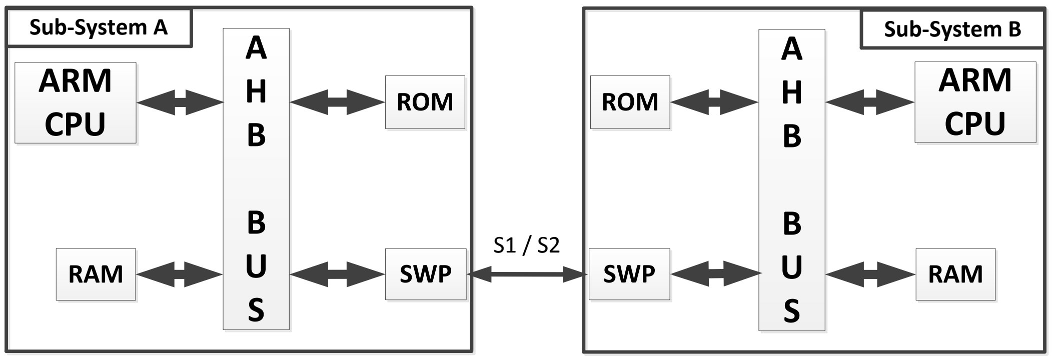


Figure 3.7: System integration of the SWP module into the overall system

Generalizations

The only LLC frame type supported is the SHDLC one. ACT and CLT frame types are not relevant for the communication layer model. The first does not need to be considered as the activation is performed in parallel to the external communication interface activation and thus won't introduce an additional delay. The latter is not modelled due to the external interface protocol overhead is already removed by the sub-system A and not forwarded to sub-system B.

Assumptions

- Not functionally implemented as considered as not necessary for on-chip communication evaluation since it is performed in parallel:
  - The interface activation of the SWP-component.
  - SHDLC *U-Frame* - responsible for link-setup and -disconnection.
- The bit duration is assumed to be 1 μs.
- CRC calculation is assumed to take 10 clock cycles (typical value for this calculation).
- The register access delays (read *and* write since no NVM is written) is assumed to be 1 clock cycle.

3.3.2 NXP Semiconductors Inter Processor Communication

The shared-memory solution which should allow higher bitrates is similar to NXP Semiconductor's IPC approach presented in Subsection 2.3.1. Its advantages are the missing intermediate bus (no bottleneck and less delay), its circular buffer structure which allows a data exchange with less copying, its bus-width transaction width (e.g. 32 bits) and its limited hardware logic needed to ensure correct data transmission[7].
The reasons to choose this architecture - based on the shared-memory communication architectures' property-listing in Table 2.2 - are as follows (with the most important reasons being listed before less important ones):

- It has the smallest amount of layers which are introduced by the architecture. Thus, the smallest area footprint as less bus-hierarchies are introduced.
- Implicitly, the above reason includes the smallest delay in terms of bus-cycles.
- The bad scalability - as this architecture is only intended to connect *exactly two* sub-systems with each other - is negligible since the target platform (specified in Subsection 1.1.4) is also fixed.

[7] Compared to the effort needed in the SWP module to handle all the protocol-related content in the HW- module.

Architecture

The architecture of the IPC component can be seen in Figure 3.8. It shows the main components of the HW module:

Interrupt Control:
: This part of the HW module is responsible for a correct activation of the interrupt signal lines IRQ0 and IRQ1, respectively. Depending on the configuration, any received data word or only a completely successful data transmission causes the target system to be notified. Additionally, when the status register is updated and the flags buf_full or buf_empty have been set, also the corresponding sub-system is notified via its IRQ line.

Register Interface:
: The register interface represents all the registers of the IPC module. These are the status as well as the control register - used to read and control the components' status - and the memory-related registers such as the pointers to the locations in memory where to write or read the next word and the registers holding the start and end addresses of the memory.

Access Arbiter:
: This unit is used to handle simultaneous accessess of both sub-systems to the same address regions (both want to access the register interface, or the same memory region). A priority-based scheme would be used to solve these conflicts and for the memory region additionally the status will be considered (if the RX memory buffer is already full, then the access of the sub-system that wants to read data from the memory will be granted).

Buffer Pointer Management:
: Every time data is written to or read from the shared memory region, the Buffer Pointer Management block is responsible for the correct updating of the dedicated registers (the memory pointer registers pointing to the address of the next word to be read and the next word to be written to).

Memory:
: The main part of this HW-module is the shared memory used to exchange data in both directions. Due to the two directions of data exchange, this block will be split up into two seperate memory regions.

The integration of the IPC component between the two sub-systems A and B is shown in Figure 3.9. Each sub-system connects its internal bus with the dedicated port on the module, thus allowing also simultaneous access on different memory parts (both sides write or read at the same time).

Abstractions

- The two bus-connections are abstracted as two TLM sockets - one for each sub-system.

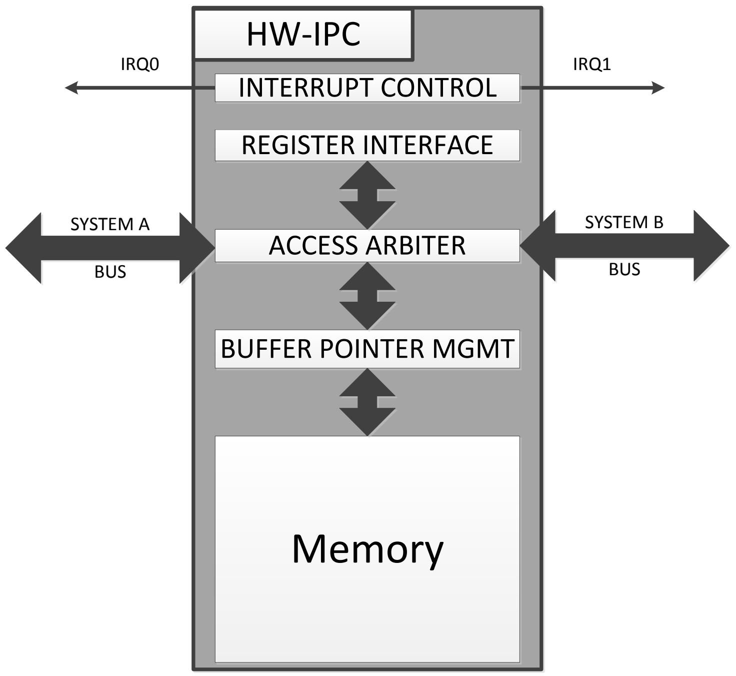


Figure 3.8: Architecture of the NXP Semiconductors IPC module showing its main parts

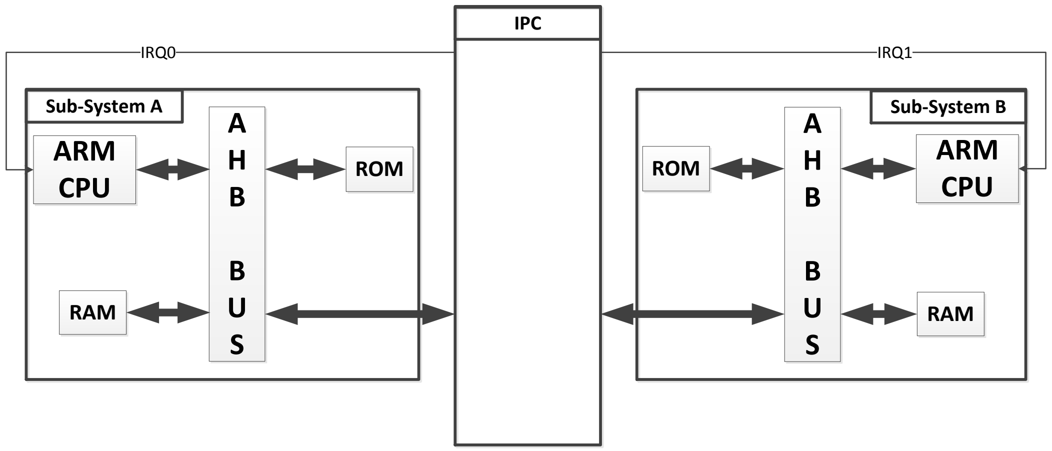


Figure 3.9: System integration of the NXP Semiconductors IPC module within the overall system

- The simultaneous-access check in the arbiter is abstracted by a timestamp-comparison of the two sub-systems' memory access. This is achieved by a context-switch caused by a *wait()* statement.

Generalizations

The bus-protocol is implemented by the corresponding HW-component and thus, no additional redundancy in form of an additional protocol is added. Data transfer errors related to the bus-transfers are thus already considered in the bus-protocol specification.

Assumptions

- The access-arbiter logic is assumed to consume one additional clock cycle due to the comparisons that have to be performed.
- Both sub-systems' clock domain is assumed to be the same - otherwise either a handshake signaling or an additional buffering-scheme would have to be introduced.
- The register read- and write-delays are assumed to be one clock cycle.

3.4 Summary

Chapter 3 begins with an overview about the simulation setup - with its synergy of the SW-applications and HW-architectures. For that purpose, the overall requirements of the evaluated applications are listed.
These contain the strict timing-response-behavior of the banking application, having a customer waiting for the application to be finished. Further, the smart metering application with its moderate amount of communication-data (a few kilobytes) but without any strict response-time limits is specified. Finally, the secure update application is defined as the challenge of exchanging a huge amount of data (hundreds of kilobytes) in combination with a user waiting for the process to be finished. For the latter two applications additionally a cryptographic-overhead has to be considered as the transactions have to be performed in a secure and authenticated manner.
Also the logging interface (SIMAPI) is presented and explained how the SW is able to log important timestamps during simulation.

This chapter further introduces the specific applications that are likely to be executed on such smartcard platforms. In order to represent different traffic-behavior these applications have been chosen to be located in different fields-of-applications. With the decision to simulate an EMV-standard based banking applet, a state-of-the-art smartcard banking application has been selected. Its setup and detailed transaction-flow is shown as well.
The other two typical smartcard applications are smart metering as well as secure software update, respectively. The first one is characterized by a verification procedure together with a rather big amount of measurement data which is sent by sub-system B to the Host. Whereas, the latter is marked by a verification procedure in conjunction with a very huge amount of data being received and stored in the local NVM.

The selected distributed-memory based communication architecture is presented in this chapter as well. SWP has been chosen first and foremost due to its very limited area consumption which is attributable to its single wire used for communication. Also its drawbacks such as the limited throughput are negligible - at least for some applications. SWP's main functional units including a detailed description thereof is presented as well as the assumptions and adaptions for the TLM2.0 compliant SystemC model.
With the selection of IPC as the shared-memory based communication approach, a very efficient solution of interconnecting two sub-systems has been presented at the end of this chapter. Its qualification is especially due to its limited layers which results in a smaller area consumption compared to the other shared-memory based architectures and a limited delay. Also the absence of a bottleneck - no intermediate bus - is another advantage. Again, the architecture of the approach is shown and a detailed description of its main parts is listed.

Chapter 4

Implementation

This chapter describes the implementation details of the application models written in *C* in Section 4.2 as well as the communication architecture models written in *SystemC* in Section 4.3.
As already mentioned in Subsection 2.1.3, the used setup of [WDL$^+$05] suits perfectly for an evaluation of different communication architectures on the communication layer. The similar simulation setup used for this work's evaluations together with other implementation specific considerations is presented in detail in Section 4.1.

4.1 Implementation Considerations

Since this work is about an evalution of several applications running on specific communication architecture approaches, there are also several parts the simulation setup consists of. The HW part is modeled in *SystemC* and LISA simulating the same timing behavior as the simulated component and the SW part - the applications - running on each subsystem's microprocessor implemented in *C*.
How all these single parts are integrated into one system, how the configuration is done and the timing results are received is shown in Subsection 4.1.1.
All the implementation specific considerations regarding the SW- and the HW-part are explained in Subsections 4.1.2 and 4.1.3, respectively.

4.1.1 Simulation Setup

The overall simulation setup for one sub-system is shown in Figure 4.1. Each sub-system consists of the same HW modules: the CPU which is modeled by ARMs *Fastmodels* (described in Subsection 4.1.3 in more detail) and the other components that are modeled as TLM 2.0 compliant SystemC models. The configuration of all these modules is done via the SystemC Modeling Library (SCML) property file which is loaded by an SCML-server that configures the single HW modules. Therefore, each sub-system and also every single HW entity can be configured independently.
Both sub-systems are integrated into one *Microsoft Visual Studio 2008* solution, containing a top-level module which consists of the two instantiations for each sub-system and their corresponding configuration summarized in the property file. The latter contains the whole HW configuration such as the clock frequency of the microprocessors and buses, the RAM

and ROM sizes, the read- and write-delays of the registers, the image file that has to be loaded and executed by the microprocessors, etc.
The mentioned binary images that are executed on each sub-system's CPU implement the embedded applications with a seperate image for each sub-system as the tasks for each of them differ depending on the underlying application. These images are *.axf* files which are cross-compiled ANSI-*C* code files using ARM's MDK - consisting of the *Keil μVision* development environment, the corresponding *Compiler* and *Linker*.
The applications are able to print the actual timestamp at very interesting points of execution (e.g. before and after a specific command is sent) via a so-called *SIMAPI* or Simulation API component. This component can be addressed by each sub-system similar to a normal memory access, however no time-delay is added since it is this only a debug-interface command.

4.1.2 Benchmark Setup

Version Information

The executables have been generated with ARMs' *KEIL μVision V4.73.0.0*, Toolchain: MDK-ARM *Standard*, C Compiler: *Armcc.exe V5.03.0.76, Assembler: Armasm.exe V5.-03.0.76, Linker/Locator: ArmLink.exe V5.03.0.76, Librarian: ArmAr.exe V5.03.0.76,* CPU Dynamic Link Library (DLL): *SARMCM3.DLL V4.73.0.0, Dialog* DLL: *DARMCM1.-DLL V1.11.00.0.*

ANSI-C

The benchmarks have been written in ANSI-C language as it has multiple advantages over a higher-level language such as *Java* for example:

- As a low-level language it allows direct manipulation of registers via pointers.
- With the KEIL μVision toolchain the C code can be ported directly to the ARM processors.
- Higher-level languages such as Java need an additional layer - i.e. the JVM - to be run on, introducing some uncertainty.

Benchmark Layering

The layering scheme of the benchmarks has been introduced in Subsection 3.1.3. Their interaction on implementation level is illustrated in Figure 4.2, with the three most important HAL interface functions the applications can use to be described as follows:

`unsigned char HalInit(void)`

This function initializes all the hardware modules being present in the current system setup. So the *init()* firmware functions of all attached modules are called which are implemented on the *Firmware Layer*. An error code is returned by the function to notify the application whether an error has occurred and if yes - which module's init() has failed.

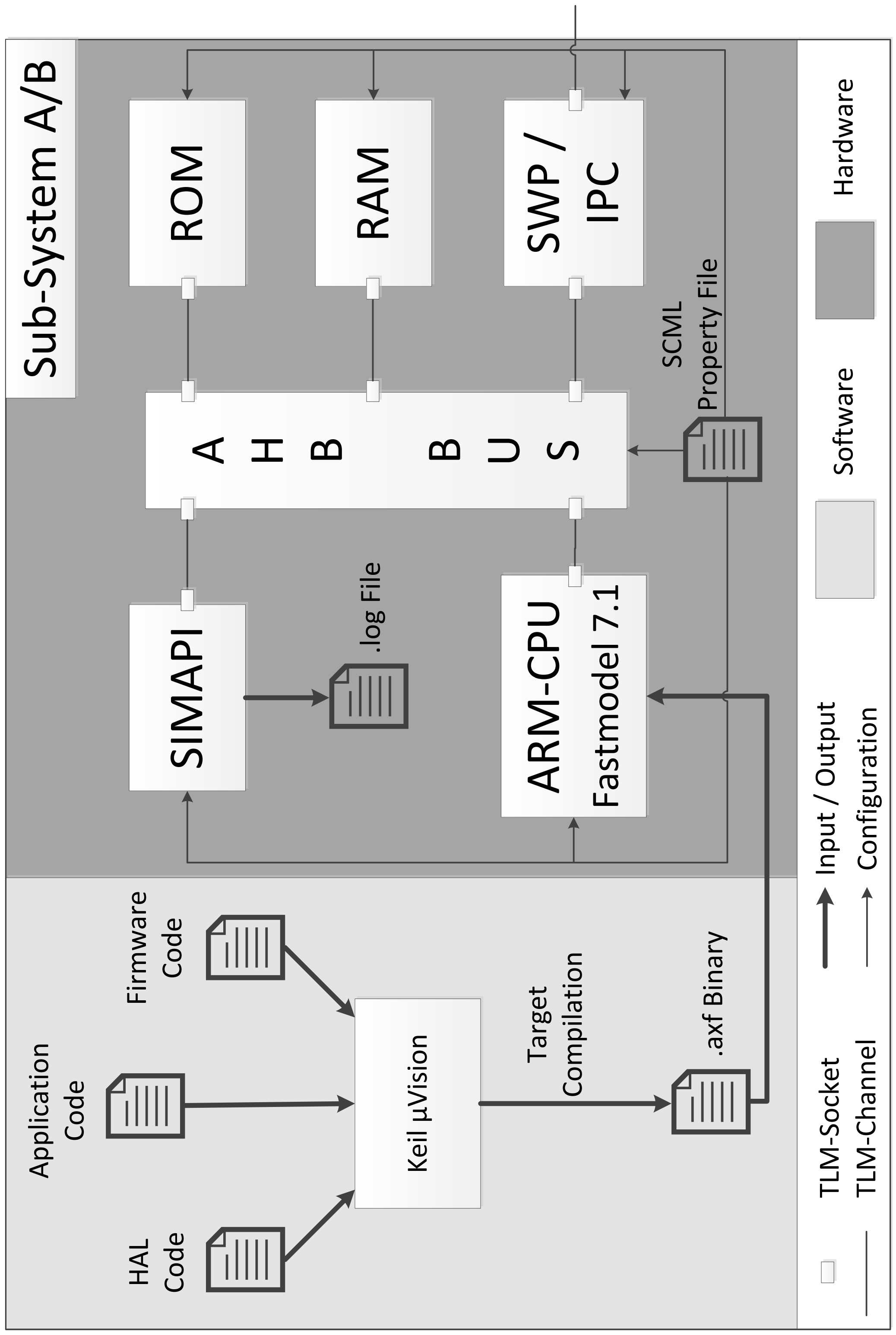


Figure 4.1: Overview of the simulation's setup implementation for one sub-system

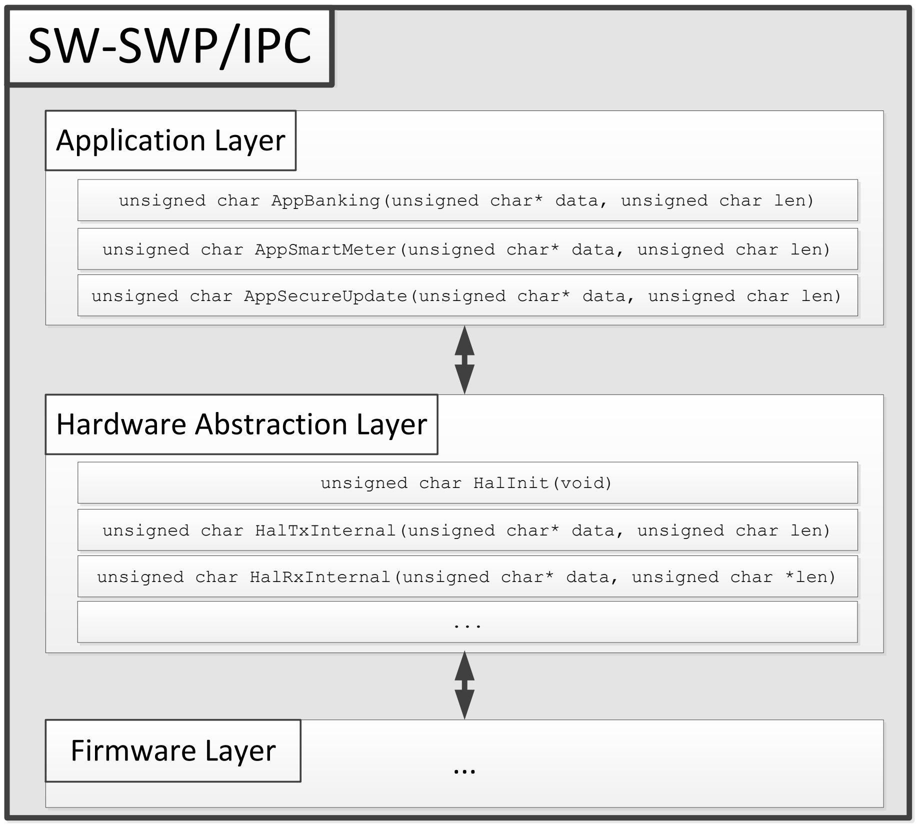

SW-SWP/IPC
Application Layer
unsigned char AppBanking(unsigned char* data, unsigned char len)
unsigned char AppSmartMeter(unsigned char* data, unsigned char len)
unsigned char AppSecureUpdate(unsigned char* data, unsigned char len)
Hardware Abstraction Layer
unsigned char HalInit(void)
unsigned char HalTxInternal(unsigned char* data, unsigned char len)
unsigned char HalRxInternal(unsigned char* data, unsigned char *len)
...
Firmware Layer
...

Figure 4.2: Interaction of benchmark layers on implementation level

`unsigned char HalTxInternal(unsigned char* data, unsigned int length)`

To transmit data from one sub-system to the other, this function can be called by the application. The parameter is a byte-array holding the data to be sent and its corresponding length. Again, an error number is returned which is zero in case no error has occurred or non-zero otherwise.

`unsigned char HalRxInternal(unsigned char* data, unsigned int* length)`

Data is received from the other sub-system via this function. The address to the byte-array reserved for the data is given as parameter as well as a pointer to the length variable. The latter has two functions: it states the length of the data-structure (how much memory has been reserved for it) to prevent a buffer overflow and contains the actual length of received data bytes when the function returns. The return-value is again the error number.

These functions then call the HW-specific functions located on the Firmware Layer. Thus, for the SWP communication architecture their interface functions are called which have been implemented with the same signature.
The *Hardware Layer* itself consists of the SystemC description of the single HW modules whose registers are set through the firmware layers' interface functions.

4.1.3 Modules Setup

Version Information

The microprocessor models as well as all the other HW models are consolidated in an *Microsoft Visual Studio 2008 Professional Version 9.0.30729.1 SP* solution with the *C/C++* toolchain information: *Microsoft Visual C++ 2008 91605-270-3439457-60499*.

ARM Fast Models

Since the used models - ARM Fastmodels 7.1 - are Instruction Accurate (IA) but not Cycle-Accurate (CA), using these models for the sole purpose of processor performance benchmarking is not adequate.
However, as the benchmarks used to evaluate the performance of the different communication architectures do not include any processor related workload being performed in the meantime but only a few register accesses for configuration of the communication module, these inaccuracies are negligible.
The *Virtual Platform* in form of a library - which is included into the Visual Studio project - is generated with the *System Canvas* tool. There the Core Fastmodel component together with the LISA- or C++-based additional components such as the *AMBAPVBus* are combined and an executable- or library-version of the Virtual Platform is built. The AMBAPVBus component is used to provide a mapping of the Advanced Microcontroller Bus Architecture (AMBA)-3 buses such as the AHB to the TLM2.0 interface.
An overview of the System Canvas setup is given in Figure 4.3.

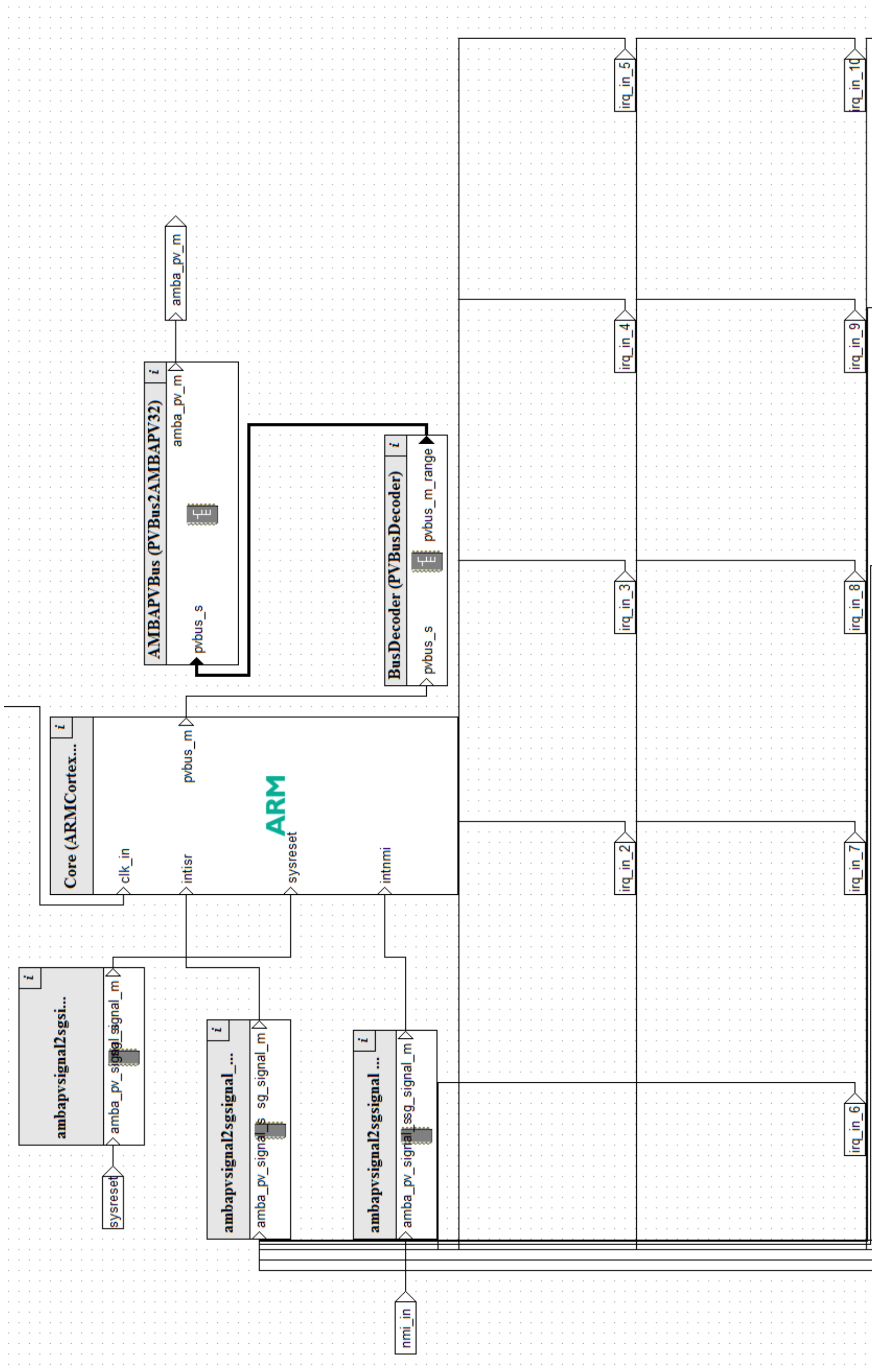


Figure 4.3: Snippet of ARM's Fastmodel implementation in System Canvas

TLM 2.0 Compliant SystemC Models

As described in Subsection 2.1.3, the abstract communication is done via so-called sockets which are a special type of a SystemC port. The function protoypes include only the transaction-related data and the timing information. The used *blocking* transport function of the TLM2 interface shown in Figure 2.9a implements the Loosely Timed (LT) version of TLM models. Into these `b_transport()` functions then the correct timing behavior related to its underlying protocol etc. is modeled. According to Subsection 2.1.3 the level of abstraction of the LT TLM models allow a proper HW architectural analysis.
All of the TLM modules being present at the company are LT modules. For the actual implementation of the specific communication architecture, also the blocking transport interface has been implemented, however additional timing points have been added at interesting points of execution. Thus, the accuracy of the HW models of the communication interface is located somewhere between LT and Approximately-Timed (AT) and can even be clock-cycle accurate. Due to the inaccuracy of the IA ISS and the early design decision phase of the communication interface evaluations, this level of accuracy is sufficient.

Simulation API

The debug Simulation API - or *SIMAPI* - component has been attached to each subsystem's bus which is responsible for the correct logging of important timestamps. Thus, this component implements the TLM interface function `b_transport(transaction_type& trans, sc_time& t)` - which is then bound to the AHB bus - and allows to be accessed by SW during simulation in the same way as any other register access.
The main difference to a normal register access is that the parameter `t` is set to `SC_ZERO_TIME` to prevent the logging from influencing the simulation.
Then the actual timestamp is retrieved via `sc_time_stamp()` and written into a log-file which can be evaluated after simulation.

4.2 Software Application Models

The implementation of the smartcard applications - with respect to the communication architecture evaluation - can be reduced to the exact number of bytes that is transmitted. Due to the generic and well-defined benchmark designflow, the whole application can be simulated with only these numbers while still being completely adequate for the purpose of the communication evaluation. Every application-specific calculation is considered in the corresponding processing-delays for each transaction.

4.2.1 Banking

The banking application is implemented according to the design specified in Subsection 3.2.1 and its corresponding transaction-flow illustrated in Figure 3.3.
As - from a communication evaluation point of view - only the amount of bytes that are transmitted is of interest, these numbers are shown here. Table 4.1 shows the detailed listing of the single transactions for the banking application together with their corresponding size in bytes.

Transaction#	*Transaction Name*	*Internal Communication[bytes]*
1	Select AID	14
2	File Control Info	43
3	Get Processing Options	9
4	Processing Options	26
5	Read Record (SFI1, REC1)	5
6	Card Data Elements	177
7	Read Record (SFI3, REC1)	5
8	Issuer Public Key	256
9	Read Record (SFI1, REC2)	5
10	Signed Static App Data	70
11	Read Record (SFI2, REC1)	5
12	ICC Public Key (1)	256
13	Read Record (SFI4, REC1)	5
14	ICC Public Key (2)	257
15	Generate AC	35
16	App Cryptogram	176

Table 4.1: Details of the banking application's transaction flow

4.2.2 Smart Metering

This is the implementation of the smart metering application according to the design specified in Subsection 3.2.2.
In order to gain meaningful results, the application is designed according to state-of-the-art cryptography mechanisms. According to National Institute for Standards and Technic (NIST) [nis] the recommended key-sizes to use through 2030 are: 2048 bits for asymmetric algorithms and at least 112 bits of security for symmetric algorithms [BBBS07]. To be on the safe side, 2048 bits for RSA and 128 bits for Advanced Encryption Standard (AES) are used as keylengths for the smart metering application.

Measurement Data Structure

As already mentioned at the beginning of Chapter 3, no standard for smart metering exists. Thus, an own representation of the measurement data had to be designed. The data-structure holding the measurement data has to be capable of storing the *Value* itself but also the corresponding *Date*, *Time* as well as some redundancy for error-detection in form of a CRC data field. The complete structure of the measurement data containing the measured power values is mentioned in Table 4.2.

Detailed Communication Flow

For the smart metering application transmission sequence, the transaction-flow is taken from Figure 3.4 combined with the designed data structure presented in Subsection 4.2.2.

The 256 bytes result from the asymmetric cryptography algorithm RSA with its keysize of 2048 bits. The exchanged commands are assumed to fit into 32 bytes thus, double the

Measurement	*Acquisition Format*	*Size[bits]*
Value	7 digits [kWh]	24
Date	6 digits [MM:DD:YY]	17
Time	4 digits [hh:mm]	12
CRC		8
RFU		3
Total		64

Table 4.2: Measurement structure setup of the smartmeter application

Transaction#	*Transaction Name*	*Internal Communication[bytes]*
1	Request Measurement Data (MD)	256
2	Send Encrypted Symmetric Key	256
3	ACK Received Symmetric Key	32
4	Send Encrypted MD	1152
5	ACK Received MD	32
6	ACK Finished MD Request	32

Table 4.3: Details of the smartmeter application's transaction flow

symmetric-block-length. The measurement data size results from the data-structure of 64 bits or 8 bytes multiplied by 6 times per hour and 24 times per day which results in 1152 bytes. This conforms to 72 times the symmetric-block-size of 128 bits and therefore adds no additional redundancy[1].

4.2.3 Secure Software Update

The implementation details of the secure software update smartcard application according to its design specified in Subsection 3.2.3 is shown here.
Again, current state-of-the-art key-sizes are used for the hybrid cryptography scheme in order to gain meaningful results. Thus, key-sizes according to NIST through 2030 are: 2048 *bits* for asymmetric cryptography and 112 *bits* for symmetric cryptography [BBBS07]. For this application, a keylength of 2048 *bits* for RSA and 128 *bits* for AES is chosen.
Today's smartcard OS memory-footprints are in the range of a hundred kB and even more, representing the largest "applications" which are present on these systems. To show the limits of the communication architectures, the numbers have been chosen to represent also the largest possible updates that could take place on current smartcard systems.

A detailed listing of the transactions that take place according to the transaction-flow specified in Figure 3.5, together with its size in bytes is shown in Table 4.4.
The initial update request and its response in form of the encrypted symmetric-key are exchanged as 256-bytes messages according to the key-size of 2048 *bits* for RSA.
Due to [WR03] the smartcard OS sizes are in the range of 3 - 250 kB with a typical average

[1] In symmetric cryptography, always blocks of symmetric-key-size data is encrypted, leading to a padding of redundant data in case the data-size does not match the block-length.

of 64 kB. However, since the size of the update varies a lot depending on which SW has to be updated and the sizes listed in [WR03] conform to OS sizes from a few years ago, an average smartcard-OS size of 128 kB is assumed.
Finally, the Update Finished commands and acknowledgements are encrypted with the 128-bit AES and assumed to fit into two of these cipher-blocks which results in 32 *bytes*.

Transaction#	*Transaction Name*	*Internal Communication[bytes]*
1	Request Init Secure Update	256
2	Send Encrypted Symmetric Key	256
3	Send Encrypted Update Data	131072
4	ACK Received Update Data	32
5	Send Update Finished	32
6	ACK Update Finished	32

Table 4.4: Details of the secure software update application's transaction flow

4.3 Hardware Communication Architecture Models

The modules consist of the HW description as SystemC TLM model as well as the Firmware written in C which sets all the registers needed for a proper initialization, data transmission and reception. For both communication architectures the descriptions are split into Section 4.3.1 for SWP and Section 4.3.2 for IPC, respectively.

4.3.1 Single Wire Protocol

The implementation of the SWP HW module with its single registers is depicted in Figure 4.4. Due to clarity and comprehensibility reasons, only the most important registers are shown.
These are the general control and status registers of the component, *CTRL* and *STAT*, respectively. Further, the TX- and RX-buffer-related control (`BUFCTRL`) and status (`TXSTAT`, `RXSTAT`) registers. The frame-length buffers - including the communication-initiator flags `RX_DATA_READY` and the communication-finished indicators `TX_DATA_READY` - in the `TXFRL` and `RXFRL` registers, respectively. Finally, the TX- and RX-buffers itself - containing the data for transmission as well as after reception (`TXBF1` - `TXBF4` and `RXBF1` - `RXBF4`) - are shown.

SystemC TLM Model

The TLM model basically consists of three `SC_THREAD`s being responsible to start the TX and RX process:

`InitTxThread()`

This thread is responsible to start the transmission sequence of I-Frames when one or more `TX_DATA_READY` flags in the register `BUFCTRL` are set. It calls the `SwpTx()` function as long as there are ready TX-buffers. When all buffers are sent, the thread waits to be notified again.

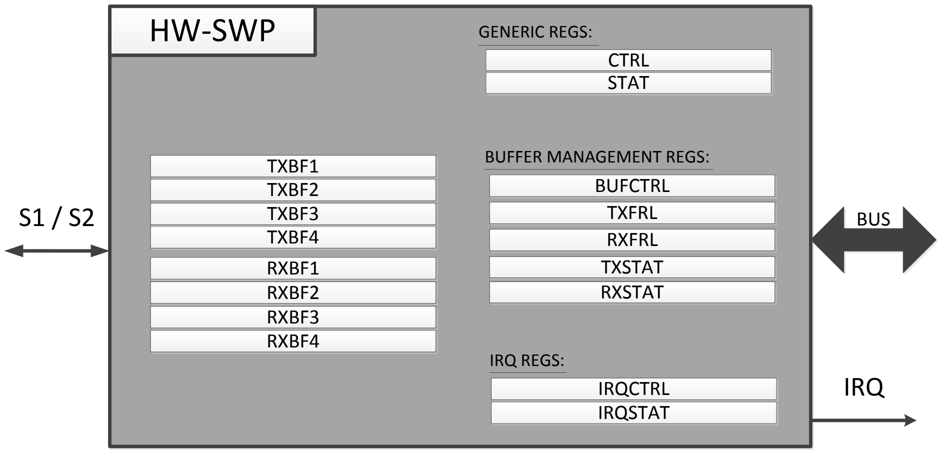


Figure 4.4: SWP implementation architecture showing its registers and interfaces

`AckTxThread()`

According to the protocol, I-Frames have to be acknowledged within the `ACKNOWLEDGE_TIMEOUT_T1`. This can be done either with an I-Frame being sent in the opposite direction, thus piggybacking the information and the acknowledgement, or with an RR-Frame to signalize the data has been received and the module is ready to receive more.
Therefore, this thread is used to initiate an acknowledge frame either immediately with an I-Frame - if data to send is ready - or after data has been received and the acknowledge-timeout has occured. Again, the function `SwpTx()` is called to perform the actual data transmission.

`EvaluateRxDataThread()`

After a new frame has been received, it will be evaluated in the function called `SwpRx()`. Depending on the frame type (I- or RR-Frame), different actions are performed in this function.

The two mentioned functions are implemented as follows:

`SwpTx()`

This function calculates the SHDLC layer content together with its related delay first. This includes a check whether the current TX-buffer is ready to be sent (I-Frame) or not (RR-Frame), setting the buffer locks, performing the sliding window check, updating the current TX-buffer and other registers according to the actual status.
Afterwards, the MAC layer data calculation is simulated and the proper delay is evaluated[2]. Then the delay time is consumed before the data is actually transmitted via the TLM socket.

[2] TLM transmission is always error-free, thus only the MAC layer related delay has to be calculated without really appending these data to the actual data content.

`SwpRx()`

The `SwpRx()` function works quite the other way around: first the MAC layer data of the frame is evaluated[3] and afterwards the SHDLC layer part. The latter includes a check whether the frame is an I-Frame or an RR-Frame.
For either frame type, the sliding window check is performed, all frames up to the current frame number are acknowledged and the next expected frame number is updated.
If it is an I-Frame, the module is set into `RX_INPROG`, the corresponding RX-buffer is locked, the data is saved and the `RX_DATA_READY` flag for the current buffer is set. Also the interrupt signal is set high after the reception of an I-Frame.
For an RR-Frame only the `RX_INPROG` is unset and the interrupt line is activated.

Besides some other - minor important register update activities - this is the main functionality of the SWP module.

Firmware Software

As it has been described in Subsection 4.1.2, the HAL interface functions call the specific implementation functions on the firmware layer. This is illustrated in Figure 4.5. The implemented firmware layer functions are as follows:

`unsigned char FwSwpInit(void)`

First of all, a so-called *Soft-Reset* is performed which flushes all TX- and RX-buffers, sets all counters (e.g. SHDLC related frame-counters) to zero, etc.
Additionally, the SWP module is set to the correct configuration. This includes the selection of the *AUTOACK* functionality[4], the `TX_AUTO_NR_E` flag is set which enables the HW to compute the receiving-frame-number automatically[5] and the interrupt is enabled for every received frame.
The return value is zero in case no error has occurred or non-zero otherwise.

`unsigned char FwSwpTx(unsigned char* data, unsigned int len)`

This function transmits length-parameter `len` bytes of the data array `data`. As long as the data array has bytes to transmit left, the actual TX buffer is evaluated and checked for its corresponding `TX_DATA_READY` to be unset, then the buffer is filled - with a maximum of 29 bytes according to the protocol - and afterwards the `TX_DATA_READY` together with the frame length is set in the `TXFRL`-register.
The return value is zero in case no error has occurred or non-zero otherwise.

`unsigned char FwSwpRx(unsigned char* data, unsigned int* len)`

The receiving function performs a check whether the actual RX-buffer's `RX_DATA_-READY` flag has been set and gets the frame's payload length via the corresponding `RXFRL`-register.

[3] Also at the RX side this is only simulated and the corresponding delay is evaluated.
[4] Automatic generation of an RR-frame after an acknowledge timeout when there is no I-frame to send.
[5] This is used in conjunction with the AUTOACK functionality to prevent possible errors due to a miss-timed SW calculation caused by high CPU load.

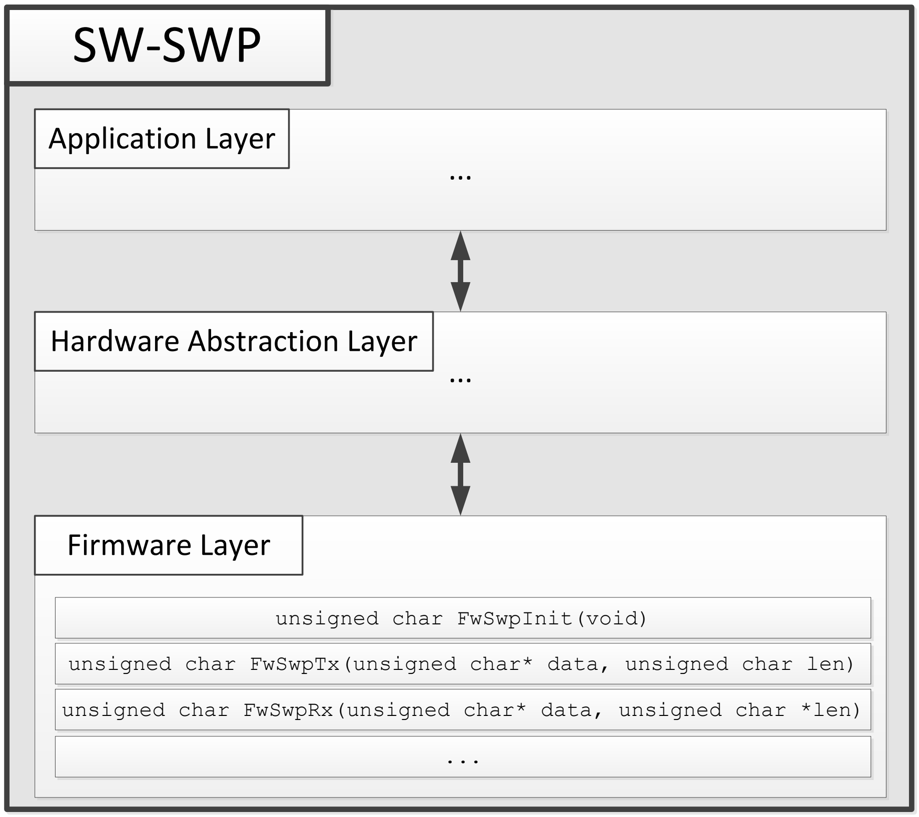

SW-SWP
Application Layer
...
Hardware Abstraction Layer
...
Firmware Layer
unsigned char FwSwpInit(void)
unsigned char FwSwpTx(unsigned char* data, unsigned char len)
unsigned char FwSwpRx(unsigned char* data, unsigned char *len)
...

Figure 4.5: Interaction of SWP's firmware layer on implementation level

If the corresponding buffer's `RX_DATA_READY` flag is not set, an RR-frame has been received, thus the length parameter `len` is set to zero. Otherwise, the length obtained by the `RXFRL`-register is compared to `len` to be sure the data array `data` is large enough and - if this is the case - it is filled with the data. Finally, the `RX_DATA_READY` is unset which frees the actual RX-buffer for another data reception.
The return value is zero in case no error has occurred or non-zero otherwise.

`unsigned char FwSwpUnsetIrq(void)`

This function is used by the application's interrupt handler to unset the IRQ signal once set by the SWP module.
It simply reads out the `IRQSTAT`-register, inverts the `RX_IRQ_STATUS` flag and writes back the value into the `IRQCTRL` register. At the end, the `IRQSTAT` register is read and evaluated again to check whether it has changed properly.
Depending on that, either zero is returned in case it has changed or non-zero otherwise.

4.3.2 NXP Semiconductors Inter Processor Communication

In Figure 4.6 the implementation architecture of the IPC approach is shown. The module consists of the status (`STAT`) and the control (`CTRL`) registers to control and read the actual state of the HW module. The memory-regions `A2B_Mem` for the data exchange from sub-system A to sub-system B. The registers for a proper synchronization of the circular-buffer-style memory `A2B_Write`, `A2B_Read` holding the next-write and next-read address in the memory block `A2B_Mem`, and the `A2B_Start` and `A2B_End` registers containing the start- and end-address of the `A2B_Mem` memory location.
The `B2A_`-registers and -memory conform to the same scheme but in the opposite direction, thus from sub-system B to sub-system A.

SystemC TLM Model

Due to the setup of the IPC component shown in Figure 3.8 with its double-bus-connection - one for each sub-system - the most important part of the HW description is placed inside the TLM2 b_transport() functions which are bound to the target socket:

`void b_transportA(transaction_type& trans, sc_time& t)`

This function implements how the HW reacts on incoming bus-requests from sub-system A. First of all, the address of the transaction parameter `trans` is evaluated. Then it is checked whether sub-system A is allowed to write or read from this address space[6]. If it is an `A2B_Mem` write access, it is checked whether the memory is full or not. Similarly for a read access on `B2A_Mem` an empty buffer check is performed. In the former case, sub-system A is notified via its IRQ signal and in the latter case sub-system B. This allows the firmware layer's functions TX and RX to react on these events, thus either wait for a non-full or a non-empty buffer, respectively.

[6] Sub-system A is only allowed to write on `A2B_Mem` and the control register `CTRL`, and it is only allowed to read from `B2A_Mem` and the status register `STAT`.

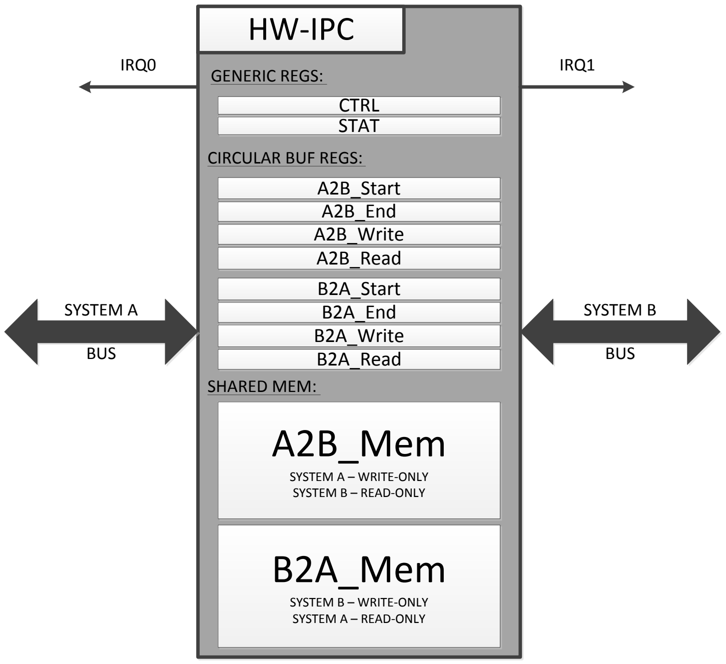


Figure 4.6: Architecture of the NXP Semiconductors IPC implementation with its shared-memory and registers

According to the access type (read or write) the proper constants for a read- or write-delay are added to the parameter `t` and the dedicated pointer registers are updated, pointing to the next read- or write-address. Additionally, for a write access the IRQ line is set[7] if the interrupt is enabled in the Status register.

`void b_transportB(transaction_type& trans, sc_time& t)`

This function implements how the HW reacts on incoming bus-requests from sub-system B. The behavior of this function conforms to the behavior of the `void b_transportA()` function however, the access scheme is vice versa. Thus `B2A_Mem` is allowed to be written to and `A2B_Mem` is allowed to be read from.

To prevent a simultaneous access of both sub-systems on the same memory region the timestamp of the access is saved and a wait() statement is executed to force a context-switch. A possible simultaneous access would then lead to the other sub-system contin-ueing its b_transport() function at the same simulation time and this would lead to one sub-system being declined[8].

[7] For a write access of `A2B_Mem` sub-system B is notified via IRQ1 and vice versa.

[8] According to some priority scheme for each memory region, combined with the status of the flags `buffer_full` and `buffer_empty` depending on whether it is a read or write access.

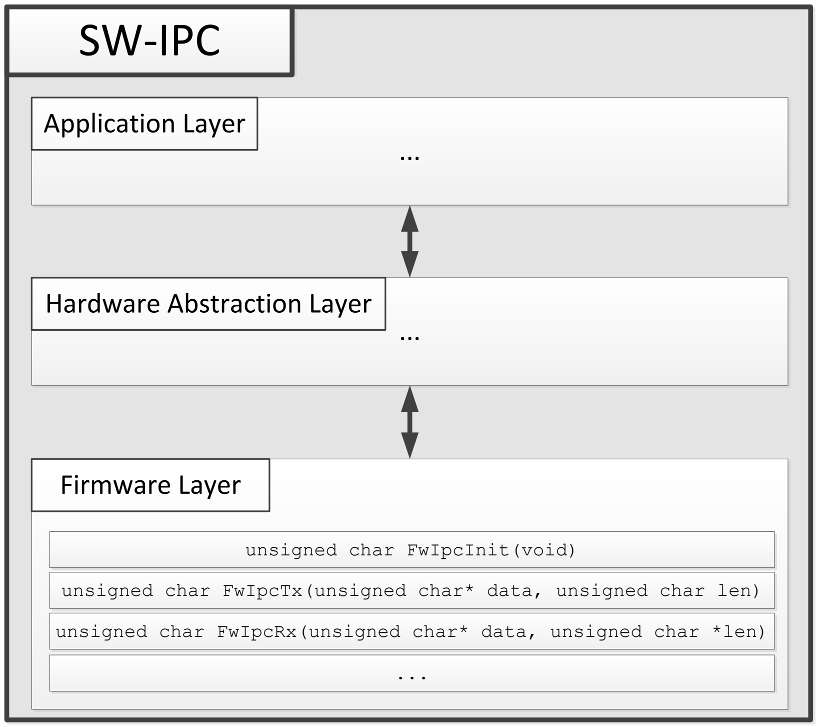

SW-IPC
Application Layer
...
Hardware Abstraction Layer
...
Firmware Layer
unsigned char FwIpcInit(void)
unsigned char FwIpcTx(unsigned char* data, unsigned char len)
unsigned char FwIpcRx(unsigned char* data, unsigned char *len)
...

Figure 4.7: Interaction of NXP Semiconductors IPC firmware layer on implementation level

The most important `SC_THREAD` of the IPC approach is the `CtrlRegChanged-Thread()`. It is responsible for calling the proper methods such as the `ResetHandler-Method()` and enabling or disabling the interrupt for received data or clearing the interrupt signal when set.

Firmware Software

Again, the firmware layer interface functions implement the HAL interface functions for the IPC approach as depicted in Figure 4.7. The interface functions can be summarized as follows:

`unsigned char FwIpcInit(void)`

In the initialization function a soft-reset is performed which includes a reset of the memory-pointer registers as well as of the memory-regions `A2B_Mem` and `B2A_Mem`. Also the interrupt is enabled for both sides. Finally, the fixed start- and end-addresses of each of the memory regions is loaded for a faster access inside each of the sub-systems.
In the event of no error has occurred for each of the register write- and read-accesses, zero is returned by the function. Otherwise, a non-zero error-code is returned.

`unsigned char FwIpcTx(unsigned char* data, unsigned int len)`

In the transmitting function, first of all the number of bytes to be sent (stored in the parameter `len`) is checked. If it is more than the system's word-size - which means multiple bus-transactions would be needed - the IRQ of the target sub-system is turned off. This is done to prevent the other sub-system to be interrupted for each word being ready to receive.
Afterwards, data is written into the buffer as long as there are data words left and no `buffer_full` flag is set. This flag is set by the interrupt handler routine which checks the flags on every interrupt.
At the end, the IRQ for the target sub-system is reactivated again, if it has been disabled in the first place.

`unsigned char FwIpcRx(unsigned char* data, unsigned int* len)`

At the beginning, the receiving function checks the length parameter `len` of the data array `data` to be a multiple of the system's word size. Then, as long as the data array is not full, a read request is performed. However, when the `buf_empty` flag has been set, the loop terminates, the length pointer `len` is updated to the actual number of bytes received and depending on whether an error occurred or not, a non-zero code is returned or a zero.

`unsigned char FwIpcUnsetIrq(void)`

This firmware function is used to clear the interrupt signal when set. Therefore, the corresponding flag of the control register (CTRL) is set which triggers the `CtrlRegChangedThread()` and finally unsets the interrupt signal. Usually, this function is called from within the interrupt handler. Thus, after the interrupt has been set by the corresponding HW, the interrupt handler is called which deactivates the non-maskable interrupts for the microprocessor, unsets the interrupt signal and reactivates the interrupt immediately afterwards.

`unsigned char FwIpcUpdateFlags(void)`

With the `FwIpcUpdateFlags()` function the internal flags corresponding to IPC's flags `buf_empty` and `buf_full` of the status register (`STAT`) are updated according to the actual status. This is important after an interrupt has occurred, since this could have been caused by either an error, received data or due to a buffer-full or buffer-empty situation. In the latter two cases, the internal flag representations `buf_full` and `buf_empty` are updated and - depending on which of these flags has been set - either the `FwIpcTx()` routine is suspended or the `FwIpcRx()` function is finished due to the empty buffer.

The interrupt handler (`IRQ_HANDLER()`) is responsible for the correct update of the sub-system's internal flags (`buf_full` and `buf_empty`) according to IPC's status register. Thus, every time one of these flags is set by the HW module, it notifies the corresponding sub-system via the interrupt signal to update its internal flags.

4.4 Summary

Chapter 4 starts with a detailed description of the overall simulation setup - shown in Figure 4.1. There, the relationship between the individual HW modules (implemented as TLM2.0 models) and the CPU (implemented as ARM Fastmodel ISS) is explained. Also the generation of the SW which is executed on the ISS as .axf image from the various layers' code files is depicted.
Afterwards, the layering scheme of the benchmark implementations is listed in detail, thus also explaining how the application-level benchmark uses the HW-specific firmware-layer functions via the HW-independent HAL interface functions which lets the HW become exchangeable.
Finally, ARM's Fastmodels 7.1. are introduced together with the way they work in conjunction with the TLM2.0 compliant SystemC models as well as their overall accuracy is stated.

In the application-models section the listing of the transaction flow extended by the exact number-of-bytes for each application is shown. For the banking application these numbers are based on the EMV contactless protocol [emv]. The accomplishment of the numbers of the smart metering and the secure software update application, however, are explained in detail together with the used encryption- and decryption-scheme and the setup of the smart meter measurement data.

The last section gives an introduction of the implementation architecture of the communication approaches, thus SWP and IPC. For each architecture, the single registers and their influence on the HW module as well as the most important functions and threads of the TLM2.0 implementation are explained.

Chapter 5

Results

This chapter lists the simulation results obtained by simulating the smartcard SW applications - described in Section 4.2 - on the HW communication architectures explained in Section 4.3. The used simulation setup is illustrated in Figure 4.1.
Due to this work's goal to evaluate the best communication architecture for a specific smartcard application, Chapter 5 is structured as follows: Section 5.2 presents the simulation results for the *banking* application. The results for the *smart metering* application are listed in Section 5.3. Finally, the *secure software update* application's simulation results is shown in Section 5.4.
Each of these sections is further divided into two subsections - conforming to the results of the SWP- as well as the IPC-communication architecture - to allow an easy comparison of the investigated architectures.
However, first of all the general simulation considerations are discussed in Section 5.1. Thus, the general parameters are stated and an example simulation flow is given for both of the considered architectures.

5.1 Simulation Considerations

It is obvious that both evaluated communication architectures have various parameters which could be altered during the simulations to detect their influence on the overall application's runtime. However, on this level of abstraction the key-parameters that qualify a specific architecture most should be evaluated first to allow an early design decision on which architecture fits best for which application.
Therefore, the fixed parameters as well as the changing parameters for the performed simulations are stated in Subsection 5.1.1 for SWP and in Subsection 5.1.2 for IPC, respectively.
The fixed parameter for both architectures are the memory and register read- and write-delays of 1 clock cycle.
The result-table's columns can be interpreted as follows:

Simulation #:
: The simulation index indicating the number of simulation performed for the corresponding communication architecture.

Bit Duration:
The duration of one bit - describing a logical 1 or a logical 0 - for the SWP approach as shown in Figure 2.12a.

Bus CLK Freq.:
AHB's bus clock frequency used for the IPC approach. This also affects the other modules being attached to the bus such as RAM and ROM.

Internal Comm.:
The sum of all the internal communication duration, thus all transactions for a specific application between the two sub-systems.

Process. Delay:
The processing delay caused by the application-specific calculations being performed between the reader's and the smartcard's data transactions.

Comm./Process.:
The relation between the overall internal communication duration to the processing delay - this is used to clarify the overhead introduced by the communication architecture.

5.1.1 Single Wire Protocol

For the first part of SWP's simulations, the internal bus-clock frequency is varied while the bit-duration is set to a value of 500 ns and the acknowledge time is set to 30 μs. As the communication via the SWP approach does not rely very much on the sub-system's internal bus but on the single wire the sub-systems are connected with, a variation of the bus-frequency only affects the register-accesses of the component.
The most important parameter for SWP is definitely its bitrate which is typically between 2 μs and 500 ns. Therefore, for the SWP approach additional simulation-results are presented in form of a second table. Due to the progression towards smaller bitrates, a bitrate of 300 ns is also considered for a possible future bitrate value used for SWP.The acknowledge timeout has been set to a fixed value of 30 μs. This is also a typical value by the time this work has been written for this paramter considering a sliding-window value of four frames at a bitrate value of 300 ns. A higher acknowledge-timeout would of course also lead to a higher overall communication duration. However, to allow the different bitrate settings to be comparable, this value has been set to be compliant with the fastest bit-duration setting whilst not indroducing an additional delay for the slower bit-duration simulations. The fixed configuration for these simulations are the CPU- and bus-frequency set to a value of 100 MHz as this is the default setting for the considered target system.

Simulation Snippet

An example simulation snippet for the SWP communication approach for the banking application is depicted in the following paragraph.
It shows the alternating frame flow of I- and RR-Frames with two RR-Frames indicating

the end of one transaction. Between Timestamp 6 and Timestamp 7 also the sliding-window effect is shown, where two consecutive I-Frames are received by the MASTER before the SLAVE receives its RR-Frame for acknowledgement. Due to the overlap of the I- and RR-Frames, the first RR-Frame received by the SLAVE only acknowledges frames received by the MASTER up to sequence number $N(R) = 1$. Only when the second RR-Frame is received by the SLAVE, the frames up to sequence number $N(R) = 2$ are acknowledged.
Please note, the data value (except the first byte) has been masked due to security reasons.

```
...
5395480 ns Timestamp 4
  5523230 ns MASTER rcvd   I-Frame N(S)=2 N(R)=2 data=0x80cccccc
  5543840 ns MASTER rcvd  RR-Frame N(R)=2
  5573840 ns  SLAVE rcvd  RR-Frame N(R)=3
6552110 ns Timestamp 5
  6596310 ns  SLAVE rcvd   I-Frame N(S)=2 N(R)=3 data=0x00cccccc
  6616420 ns  SLAVE rcvd  RR-Frame N(R)=3
  6647420 ns MASTER rcvd  RR-Frame N(R)=3
7624760 ns Timestamp 6
  7765590 ns MASTER rcvd   I-Frame N(S)=3 N(R)=3 data=0x70cccccc
  7816200 ns  SLAVE rcvd  RR-Frame N(R)=4
  7902200 ns MASTER rcvd   I-Frame N(S)=4 N(R)=3 data=0x13cccccc
  7952810 ns  SLAVE rcvd  RR-Frame N(R)=5
  8038810 ns MASTER rcvd   I-Frame N(S)=5 N(R)=3 data=0x1acccccc
  8089420 ns  SLAVE rcvd  RR-Frame N(R)=6
  8175420 ns MASTER rcvd   I-Frame N(S)=6 N(R)=3 data=0x5fcccccc
  8226030 ns  SLAVE rcvd  RR-Frame N(R)=7
  8312030 ns MASTER rcvd   I-Frame N(S)=7 N(R)=3 data=0x26cccccc
  8362640 ns  SLAVE rcvd  RR-Frame N(R)=0
  8448640 ns MASTER rcvd   I-Frame N(S)=0 N(R)=3 data=0xf1cccccc
  8481250 ns MASTER rcvd   I-Frame N(S)=1 N(R)=3 data=0x82cccccc
  8499250 ns  SLAVE rcvd  RR-Frame N(R)=1
  8501860 ns MASTER rcvd  RR-Frame N(R)=3
  8531860 ns  SLAVE rcvd  RR-Frame N(R)=2
9509140 ns Timestamp 7
...
```

5.1.2 NXP Semiconductors Inter Processor Communication

For the IPC communication architecture approach, the most influencing parameter is the sub-system's bus clock frequency. The default value of this clock frequency is 100 MHz, however also simulations with a different clock speed have been performed.
The sizes of the memory blocks of the IPC component for each direction (`MemA2B`, `MemB2A`) are defined to be large enough to store all the data corresponding to a single transaction of each application. Thus, the whole transaction can be finished at once without the need of one sub-system's intervention to empty the buffer before the other sub-system can continue transmitting its data.

Simulation Snippet

The following snippet shows a part of communication for the banking application. Here the transactions four to seven of the banking application's transaction flow are shown. The address offset of the `MemB2A` memory block within the IPC component is $0x20300$ and the offset of the `MemA2B` memory block is $0x200$.
It can be seen, that data is transferred from one sub-system's RAM into the corresponding memory block of the IPC component (e.g. WriteMemB2A). Afterwards, the data is read by the other sub-system (e.g. ReadMemB2A) and stored within its internal RAM for further processing.
The empty buffer is then recognized by the HW module itself and the corresponding flag is set which leads to the termination of the read process.
Again, the data value is masked (except the first byte) due to security reasons.

```
...
5229840 ns Timestamp 4
  5236850 ns WriteMemB2A on address=0x2032c data=0x80cccccc
  5239260 ns WriteMemB2A on address=0x20330 data=0x80cccccc
  5242620 ns WriteMemB2A on address=0x20334 data=0x80cccccc
  ...
  5254310 ns WriteMemB2A on address=0x20344 data=0x90cccccc
  5262 us ReadMemB2A from address=0x2032c data=0x80cccccc
  5262550 ns ReadMemB2A from address=0x20330 data=0x80cccccc
  5263100 ns ReadMemB2A from address=0x20334 data=0x80cccccc
  ...
  5265300 ns ReadMemB2A from address=0x20344 data=0x90cccccc
  [IpcLT::ReadMemB2A]: Circular Buffer Empty!
6276850 ns Timestamp 5
  6282810 ns WriteMemA2B on address=0x21c data=0xb2cccccc
  6283950 ns WriteMemA2B on address=0x220 data=0x00cccccc
  6291580 ns ReadMemA2B from address=0x21c data=0xb2cccccc
  6293080 ns ReadMemA2B from address=0x220 data=0x00cccccc
  [IpcLT::ReadMemA2B]: Circular Buffer Empty!
7303540 ns Timestamp 6
  7309590 ns WriteMemB2A on address=0x20348 data=0x70cccccc
  7312960 ns WriteMemB2A on address=0x2034c data=0x66cccccc
  7316320 ns WriteMemB2A on address=0x20350 data=0x5fcccccc
  ...
  7445010 ns WriteMemB2A on address=0x203f8 data=0x00cccccc
  7453460 ns ReadMemB2A from address=0x20348 data=0x70cccccc
  7454010 ns ReadMemB2A from address=0x2034c data=0x66cccccc
  7454560 ns ReadMemB2A from address=0x20350 data=0x5fcccccc
  ...
  7483410 ns ReadMemB2A from address=0x203f8 data=0x00cccccc
  [IpcLT::ReadMemB2A]: Circular Buffer Empty!
8494950 ns Timestamp 7
...
```

5.2 Banking

The simulation results of the banking application - with its transaction flow depicted in Figure 3.3 and a detailed listing of its corresponding transaction-sizes in bytes given in Table 4.1 - are presented in this section.
The timing-values for the SWP communication approach varying the bus-clock frequency are shown in Table 5.1, whereas the values varying the bitduration parameter are shown in Table 5.2, respectively. Consequently, the simulation-runtime values for the IPC architecture used as an interconnection-fabric are listed in Table 5.3.
The meaning of the table's columns is explained at the beginning of Section 5.1.

5.2.1 Single Wire Protocol

The banking application is defined by its complex cryptographic calculations and a rather limited amount of data being transmitted between each sub-system.
A bus-clock frequency of less than the default value of 100 MHz shrinks the percentage of the communication overhead as the SWP-related communication overhead is hardly affected by the bus-clock frequency. However, due to the high computation-effort of the banking application, a slower clock speed leads to an overall simulation time of far more than the required 400 ms for the smartcard alone according to [mas]. Therefore, the SWP communication architecture is also considered with different bit-duration settings. With a bit-duration of 500 ns it only adds $2,47\%$ to the overall execution time compared to the processing duration. This is due to the overall communication overhead of just 1344 bytes being transmitted between the two sub-systems whilst having a big processing delay of nearly 285 ms. Even the very slow bit duration of 2 μs does not even add 10% in comparison to the computation overhead.
A possible future configuration of 300 ns for the bit duration would even lead to a communication overhead of only 4,30 ms which is - compared to the 284,95 ms - almost negligible.

Simulation#	*Bus CLK Freq.* [MHz]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	$20,00$	7882,87	1424,75	0,55
2	$50,00$	7193,35	569,90	1,26
3	$66,67$	7120,63	427,43	1,67
4	$100,00$	7044,13	284,95	2,47

Table 5.1: SWP banking simulation results with varying bus-clock frequency

5.2.2 NXP Semiconductors Inter Processor Communication

IPC's strength - its high bitrate - is demonstrated for the first time when the communication delay caused by the IPC architecture is put in relation to the banking application's processing delay. With values between $0,51\%$ and $0,58\%$, the communication overhead for this application would be negligible.

Simulation#	*Bit Duration* [ns]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	2000	27570,31	284,95	9,68
2	1000	13733,97	284,95	4,82
3	500	7044,13	284,95	2,47
4	300	4303,67	284,95	1,51

Table 5.2: SWP banking simulation results varying the bitduration

Since different bus-clock frequencies are simulated - which of course influences the processing delay - the relation between communication and computation is kept on a similar level.
Due to the future requirement of EMV banking applications [mas] from 2016 onwards, an overall duration (communication and processing) of 300 ms for the smartcard alone, a slower clock-speed than 100 MHz is not an option considering actual processing-delay values.

Simulation#	*Bus CLK Freq.* [MHz]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	$20,00$	7216,82	1424,75	0,51
2	$50,00$	2966,69	569,90	0,52
3	$66,67$	2293,15	427,43	0,54
4	$100,00$	1637,14	284,95	0,58

Table 5.3: IPC banking simulation results with varying bus-clock frequency

5.3 Smart Metering

The simulation results of the smart metering application - with its transaction flow depicted in Figure 3.4 and a detailed listing of its corresponding transaction-sizes in bytes given in Table 4.3 - are presented in this section.
The timing-values for the SWP communication approach varying the bus-clock frequency are shown in Table 5.4. The resulting values for the simulation with the bitduration as parameter are presented in Table 5.5. Consequently, the runtime-values for the IPC architecture used as an interconnection-fabric are listed in Table 5.6.
The meaning of the table's columns is explained at the beginning of Section 5.1.

5.3.1 Single Wire Protocol

Due to the design of the smart metering application - which allows a lot of computation being performed in the background while the system is idle - the communication duration is high in relation to the calculation duration.
As the communication duration for the SWP approach is not affected by the bus-clock

speed, for the smart metering application with its soft timing requirements, the low clock-speed of only 20 MHz would be an option.
For the rather slow bit-duration of 2 μs, the communication overhead for the SWP architecture adds up to already more than two-thirds of the processing delay. A possible evolution of the bit-duration towards 300 ns in the near future would relativize the overhead to approximately 10%. However, with approximately 17% communication overhead in relation to the processing overhead, also the typical bit-duration value of 500 ns does not add a significant overhead compared to the computation-delay.

Simulation#	*Bus CLK Freq.* [MHz]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	20, 00	9013,31	250,78	3,59
2	50, 00	8608,48	100,31	8,58
3	66, 67	8579,48	75,23	11,40
4	100, 00	8549,99	50,16	17,05

Table 5.4: SWP smart metering simulation results with varying bus-clock frequency

Simulation#	*Bit Duration* [ns]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	2000	33947,90	50,16	67,69
2	1000	17015,90	50,16	33,93
3	500	8549,99	50,16	17,05
4	300	5161,43	50,16	10,29

Table 5.5: SWP smart metering simulation results varying the bit-duration

5.3.2 NXP Semiconductors Inter Processor Communication

For the typical setup of a bus-clock frequency of 100 MHz, the IPC communication variant would only add approximately 1,85 ms on communication overhead to a processing delay of about 50 ms.
Even for the low clock-frequency simulation run with a clock-frequency of 20 MHz, the overall simulation runtime only sums up to approximately 259 ms - which is still not very much considering that this is only done once a day and without any human interaction. However, in comparison to the low-power and low-area approach SWP which reaches a similar result for a clock-frequency of 20 MHz, a system setup with IPC configured with 20 MHz would be less efficient.

5.4 Secure Software Update

The simulation results of the secure software update application - with its transaction flow depicted in Figure 3.5 and a detailed listing of its corresponding transaction-sizes in bytes given in Table 4.4 - are presented in this section.

Simulation#	*Bus CLK Freq.* [MHz]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	$20,00$	8382,15	250,78	3,34
2	$50,00$	3331,46	100,31	3,32
3	$66,67$	2564,66	75,23	3,41
4	$100,00$	1849,19	50,16	3,69

Table 5.6: IPC smart metering simulation results with varying bus-clock frequency

The timing-values for the SWP communication approach varying the internal bus-clock frequency are shown in Table 5.7 and the simulation-results for the variation of the bit-duration are listed in Table 5.8. Consequently, the runtime-values for the IPC architecture used as an interconnection-fabric are listed in Table 5.9.
The meaning of the table's columns is explained at the beginning of Section 5.1.

5.4.1 Single Wire Protocol

With the complex cryptographic operations of the RSA-sign and -verification process being performed at the key-exchange step at the beginning of the application's transaction flow, the high processing delay of 2,39 *sec* results.
For the bit-duration setting of 2 μs, the communication overhead even exceeds the very high processing delay. This sums up to an overall duration of 4,86 *sec*. This is too much when a human being is considered to wait for the smartcard to finish this process.
Although, due to the nature of the SWP architecture, the communication overhead does not significantly increase for lower bus-clock frequencies, such a configuration is not reasonable as the overall execution-time would be far too high for a person being comfortable to wait for the process to finish.
A typical value for the bit-duration of 500 ns would still lead to an overall time of approximately 3 *sec*.

Simulation#	*Bus CLK Freq.* [MHz]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	$20,00$	619467,54	11926,35	5,19
2	$50,00$	619284,93	4770,54	12,98
3	$66,67$	619252,57	3577,91	17,31
4	$100,00$	619224,06	2385,27	25,96

Table 5.7: SWP secure update simulation results with varying bus-clock frequency

5.4.2 NXP Semiconductors Inter Processor Communication

Due to the very high processing delay that is associated with this secure update application, an IPC approach with a bus-clock being down-scaled to a clock-frequency of 20 MHz results in an overall execution time of approximately 12,5 *sec* - with the external interface and the reader-related delay not even considered.

Simulation#	*Bit Duration* [ns]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	2000	2471485,57	2385,27	103,62
2	1000	1236954,29	2385,27	51,86
3	500	619224,06	2385,27	25,96
4	300	372132,08	2385,27	15,60

Table 5.8: SWP secure update simulation results varying the bit-duration

However, a setting of 100 MHz for the bus-clock leads to an overall delay of just 2,5 *sec*, which is an acceptable value when the huge amount of data that is being transmitted is kept in mind.

Simulation#	*Bus CLK Freq.* [MHz]	*Internal Comm.* [μs]	*Process. Delay* [ms]	*Comm./Process.* [%]
1	$20,00$	524122,93	11926,35	4,40
2	$50,00$	223689,87	4770,54	4,69
3	$66,67$	172993,41	3577,91	4,84
4	$100,00$	123310,04	2385,27	5,17

Table 5.9: IPC secure software update simulation results with varying bus-clock frequency

5.5 Summary

In the first section of Chapter 5 the general considerations regarding the simulations are presented. This includes the parameters that have been fixed for all simulations as well as the parameters that have been changed for each communication architecture in the single simulations.
Also the result-table's columns are explained and their influence on the overall simulation runtime for each of the communication architectures is stated.
The example simulation snippet - showing a part of the banking application for both of the communication approaches - is used to give a short overview about the performed simulations and results.

After this, the banking application's results are presented. Due to the limited communication traffic this application is characterized by, the resulting communication overhead stays within short limits even for the slower SWP approach. For a typical configuration of 100 MHz for the bus-clock frequency, a bit-duration of 500 ns and an acknowledge-timeout value of 30 μs, the resulting internal-communication overhead is only $3,69\%$.

The smart metering application is marked by its little processing delay. This is due to the nature of this application where a huge part of the calculations can be performed between two consecutive measurement data requests.
Together with the moderate traffic behavior, the differences between the two communica-

tion architecture approaches become apparent, where the fastest simulation result of the SWP approach sums up to 5,16 ms while the IPC architecture results in only 1,85 ms. Due to the soft timing-limit of this application, even a down-scaling of the internal bus-clock frequency to 20 MHz would make sense, resulting in an overhead of only $3,59\%$ related to the computation-delay (for a bit-duration of 500 ns).

Finally, the secure software update's results are presented. Due to the large amount of data being transmitted - the software-update size is assumed to be 128 kB - the strenghts of the IPC approach become apparent.
For the typical bit-duration setting of 500 ns for the SWP architecture, the overall communication duration sums up to 619,22 ms. Although this communication overhead in absolute numbers is rather big for the IPC as well, with an overall duration of 123,31 ms it is nearly half a second faster.
A system configuration with an internal bus-speed of 20 MHz is absolutely pointless for this application for both considered communication architectures resulting in an overall execution-time of approximately 12,5 sec.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The work's goal is the evaluation of the influence of different on-chip communication architectures for specific smartcard-application's execution time. The target applications are: a typical banking application with a small communication-traffic behavior, a rather big processing-delay related to its cryptographic calculations and a strict timing-limit according to its specification. A smart metering application with a moderate communication behavior, little computation overhead to to the application's scenario allowing the main part being performed between two consecutive requests and basically no timing limit. As well as a secure software-update application with a huge communication profile due to today's smartcard-OS-sizes, a high processing delay related to the very-complex, asynchronous cryptography based verification process and a timing limit due to the human interaction when holding the smartcard into a reader's field for example.

Hardware and Software Classification

In order to allow a rather fast evaluation of different architectures and applications, the right level of abstraction for the simulated HW and SW had to be found first. The SystemC version 2.2 with its library extension TLM2.0 is chosen to model the abstracted HW communication models. The SW is implemented in ANSI-C and is executed on ARM's Fastmodels 7.1 ISS.
Based on Flynn's classification for MIMD systems, the considered communication architectures are divided into distributed-memory- as well as shared-memory-based architectures. The former led to an implementation of the SWP approach with very limited area needs as well as power requirements due to its single wire architecture. Whereas the latter led to the bus-based IPC communication architecture with a rather high data-throughput but introducing an additional area-overhead at the same time.

Abstraction Aspects

Due to the nature of smartcard-applications - always, a command is sent from the reader to the smartcard and a response is sent back - a generic benchmark solution has been introduced, allowing an accurate modelling of the application whilst keeping the design-overhead small. With the application's processing delay being considered inside this benchmark, a possible influence on the investigated communication architecture is captured. The HW models describing the corresponding interconnection-fabric are designed to consider any protocol-related communication-overhead - resulting in a timing delay. However, due to the high abstraction layer, the computational overhead is kept on an acceptable level.

Architecture Decision

Each of the specific smartcard-application's requirements together with the simulation results allow a proper choice of which communication architecture is more appropriate for which application:

A typical banking application according to [mas] has to comply to an overall runtime of 500 ms until 2016, with 400 ms for the smartcard alone. With slightly more than 7 ms conforming to the internal-communication overhead caused by SWP - for the configuration with a bit-duration of 500 ns - the performance of the communication architecture is sufficient. Although, the IPC variant only conforms to a communication duration of approximately 1,64 ms - for the typical configuration of a bus-clock frequency of 100 MHz - the limited influence on the overall performance as well as the much higher area needs and power requirements do not justify this approach.
However, as the banking application's specification from until 2016 [mas] requires an overall execution duration of only 400 ms while the cryptographic-overhead will not shrink too significantly due to the rising key-sizes, any potential savings with respect to the delay are to be considered. Possible solutions for this would be either the usage of a smaller bit-duration for the SWP architecture or a higher-bandwidth solution.

The moderate communication profile of the smart metering application together with its very soft timing-restrictions - the data is requested only once per day - allow the usage of a low-power design - consisting of an SWP communication architecture with a bit-duration of 500 ns, an acknowledge time of 30 μs and an internal bus-clock frequency of only 20 MHz - resulting in an overall execution time of less than 260 ms. Due to this application's scenario of a host requesting the data - without any user-interaction - a faster communication architecture correlating with a higher power consumption as well, won't be adequate.

A secure software update application allowing the user or customer to update a specific part of the smartcard-SW results in huge amounts of data being transmitted between the reader and the smartcard. With a high processing delay due to a complex asynchronous-cryptography-based authentication scheme and a human waiting for the whole process to finish, this scenario definitely justifies the usage of a higher-bandwidth on-chip communication solution such as IPC.
A communication-overhead of about 123 ms for the IPC approach with a bus-clock frequency of 100 MHz compared to the nearly 620 ms for the SWP approach for a bit-duration configuration of 500 ns clarifies the difference of these communication approaches. Even the possible future bitrate of 300 ns - which SWP is not capable of by the time this

work has been written - would reduce the internal communication overhead by only a third to approximately 372 *ms*. Thus, for the secure software update scenario as described in this work with its hybrid cryptography scheme and an update of a large SW such as a smartcard OS, the IPC communication architecture would be appropriate.

6.2 Future Work

In the course of this work three smartcard applications are evaluated on two different communication architectures. This work's goal was to find a communication architecture that fits best for a specific application. The selected applications are chosen according to their different communication traffic-behavior covering a lot of the smartcard-applications' traffic scenarios being present at the moment.
Apparently, as it could be seen in Section 6.1, the selection of a specific communication architecture for a specific smartcard application is highly application-dependent - with its numerous properties and restrictions that are related to it. Thus, a categorization of applications into groups of similar traffic-behavior alone is not enough as - for an early design decision - the application's requirements have to be considered as well. Therefore, a detailed evaluation of different communication architectures for each smartcard application that is considered to be used in a system is inevitable.
Due to the generic benchmark solution presented in Subsection 3.1.3 a big variety of applications can be evaluated easily for existing communication architecture models. Bearing the proper requirements and properties related to each application in mind, an early design decision on which architecture to use for which target application can be performed directly after the high-level simulations.

As already mentioned in Subsection 1.1.1, embedded systems have numerous requirements and restrictions to meet. One of them is especially important for smartcard systems: power. Due to the importance of this property and its influence on the design decision whether to use a specific communication architecture or not, a possible future work could be the enhancement of the HW architecture models by a high-level power-estimation that works on this rather abstract level such as the SystemC class library *Powersim*[GOC11].

Another aspect for a possible future work is an enhancement of the HW models' accuracy. If these models will be integrated into a cycle-accurate simulation model for a detailed performance analysis, their TLM2.0 interface should be enhanced to implement a cycle-accurate, non-blocking (`nb_transport()`) interface function. With the benefits of more detailed simulation-models already discussed in Subsection 2.1.2, also inaccuracies caused by possible bus-congestions due to multi-master accesses can be detected accurately. Of course, also the CPU model - the ISS - has to support cycle-accuracy in order to perform such accurate performance evaluations, allowing also the detection of e.g. cache-misses.
Of course, on this high level of detail, also the generic benchmark solution has to be adapted. In order to be able to model even cycle-accuracy-related delays, the single HW components being involved in the calculations would have to be simulated seperatedly. This would then also allow the detection of delays related to a possible blocking of these components among each other or through the interconnection-bus.

With SWP, a state-of-the-art smartcard communication architecture has been selected for evaluation. As this architecture is typically already present in current smartcard systems, this approach serves more as a reference for a competing intercommunication-fabric. The evaluation of IPC as a on-chip smartcard communication infrastructure - nowadays, using a bus-based interconnect approach on smartcards is rather unconventional - shows a possible way towards the evolution of smartcards' communication architecture. As soon as the power dissipation as well as the area needs of these bus-based approaches can be limited to a reasonable factor, these setups could become typical for mid-term smartcard solutions. This is related to the higher scalability of such bus-based communication infrastructure allowing them to connect more sub-systems easily[1] and the increasing demand of smartcard applications in terms of their memory footprint as well as communication traffic behavior. Therefore, additional bus-based communication approaches - with different setups such as a hierarchical-bus scheme - should be evaluated in a possible future work. Also an IPC version with limited memory space could be evaluated, where each sub-system would have to transmit or receive data multiple times per transaction in order to empty the memory buffer in case it is full.

An interesting approach with a good trade-off between the architecture's throughput and its power-requirements could be the SPI connection scheme. For today's smartcard applications, this architecture offers enough throughput while it keeps the area-needs and power consumption on an acceptable level since only four wires are used.
For specific applications - especially where multi-master conflicts have to be solved quite often - also the low-power I^{2}C approach presented in Subsection 2.2.5 could be of interest. This architecture would be particularly interesting for applications with a rather limited communication traffic among the components - due to its limited bandwidth of 3.4 MBit/s - but multiple communication-masters affected during the processing.

On the long-term, probably NoC-based communication architectures will fit best the future smartcard applications' needs. With the ongoing miniaturization it is very likely that each self-contained task with some minimum of complexity will result in its own sub-system leading to dozens or even hundreds of sub-systems that have to be connected with each other. Such a system offers the biggest degree of flexibility - but also complexity - and would be able to process also very complex applications. Again, the power consumption and area-requirements for these NoC-based systems - due to the huge number of wires resulting from such an interconnection-scheme - has to be limited first. Due to this reasons, upcoming NoC-based architecture-approaches should also be considered in a future work evaluation.
By the time the power consumption and area-needs of such heavy-wire architectures will come to an acceptable level for passively powered devices such as smartcards, another interesting communication variant would be the IQP one. This architecture would be especially interesting for applications with a very high communication demand between various sub-systems at the same time whilst having a rather tight execution time limit at the same time.

[1] Up to a few sub-systems being added, a bus-based approach has a rather good scalability, as long as not too much data is sent simultaneously.

Appendix A

Glossary

A.1 Abbreviations

μC Micro-Controller. 7–9, 27, 39

ABS Anti-lock Braking System. 3

AC Application Cryptogram. 55

AC Alternate Current. 36

ADL Architecture Description Language. 18

AES Advanced Encryption Standard. 74–76, *Nomenclature:* AES

AFL Application File Locator. 55

AHB Advanced High-performance Bus. 41, 71, 73, 86

AID Application IDentification. 53

AIP Application Interchange Profile. 55

AMBA Advanced Microcontroller Bus Architecture. 71

ANSI American National Standards Institute. 20, 68, 95

API Application Programming Interface. 51, 52, 68, 73

ARM Advanced RISC Machines. 7, 19, 24, 40, 49, 51, 67, 68, 71, 83, 84, 95

AT Approximately-Timed. 22, 23, 73

CA Cycle-Accurate. 19, 23, 71

CLF Contact-Less-Frontend. 28, *Nomenclature:* CLF

CMOS Complementary Metal Oxide Semiconductore. *Nomenclature:* CMOS

CPHA Clock PHAse. 31

CPOL Clock POLarity. 31

CPU Central Processing Unit. I, II, 5, 7, 14, 17, 27, 40, 41, 44, 49, 67, 68, 78, 83, 86, 97

CRC Cyclic Redundancy Check. 1, 29, 31, 37, 38, 59, 60, 62, 74

DC Direct Current. 36

DLL Dynamic Link Library. 68

DMA Direct-Memory-Access. *Nomenclature:* DMA

DPCI Data Pre-fetch Core Interface. 42–45

EEPROM Electrically Erasable Programmable Read-Only Memory. 4, 5

EMV Europay international Mastercard Visa. 48, 53, 65, 84, 90

EOF End of Frame. 29, 31, 60

ESC Electronic Stability Control. 3

FCI File Control Information. 53

FSB Front Side Bus. 36, 38

FSM Finite State Machine. 44, 45, 47

GDSII Graphic Data System II. 14, *Nomenclature:* GDSII

GET Get Processing Options. 55

HAL Hardware Abstraction Layer. 51, 68, 78, 82, 84

HDL Hardware Description Language. 20, 23

HW Hardware. I, II, 2, 4, 9–13, 17, 19, 20, 22–24, 26, 35, 38–41, 45, 48, 49, 51, 59, 62, 63, 65, 67, 71, 73, 76, 78, 80, 81, 83–85, 88, 95–97, 105

I^{2}C Inter-Integrated Circuit. 38, 98

I/O Input / Output. 5, 7

IA Instruction Accurate. 19, 71, 73

IC Integrated Circuit. 4–6, 35, 39, 59

ICC Integrated Circuit Card. 55

IDE Integrated Development Environment. 49, 51

IEEE Institute of Electrical and Electronics Engineers. 35, *Nomenclature:* IEEE

IP Intellectual Property. 42

IPC Inter Processor Communication. 44, 45, 59, 62, 63, 66, 76, 80–92, 94–98

IQP Intel Quick Path. 36–38, 98

IRQ Interrupt ReQuest. 60, 63, 80, 81, 83

IS Instruction Set. 49

ISS Instruction Set Simulator. 18, 19, 24, 27, 73, 83, 84, 95, 97

JTAG Joint Test Action Group. 35, 38, 39

JVM Java Virtual Machine. 17, 68

LISA Language for Instruction Set Architecture. 18, 19, 26, 27, 49, 67, 71

LLC Logical Link Control. 29, 31, 60, 62

LPDU Link Protocol Data Units. 29

LT Loosely Timed. 22, 23, 73

MAC Medium Access Control. 29, 34, 60, 77, 78

MDK Microcontroller Development Kit. 51, 68

MIMD Multiple Instruction Multiple Data. 6, 7, 27, 95

MISD Multiple Instruction Single Data. 6

MISO Master Input, Slave Output. 31, 35

MOSI Master Output, Slave Input. 31, 35

MPSoC Multi-Processor System-on-Chip. I, II, 1, 27

NFC Near Field Communication. 3, 8, 38, 47, 57, 59

NIST National Institute for Standards and Technic. 74, 75

NoC Network-on-Chip. 25, 27, 98

NPU Numeric Processing Unit. 5

NVM Non-Volatile-Memory. 17, 18, 53, 57, 62, 65

OS Operating System. 1, 5, 12, 49, 57, 75, 76, 95, 97

PA-BCA Pin-Accurate Bus Cycle Accurate. 15, 16

PC Personal Computer. 2, 19, 36

PCB Procotol Control Byte. 34

PCI Peripheral Component Interconnect. 18, 38

PDOL Processing Options Data Object List. 53

PE Processing Element. 14

PGP Pretty Good Privacy. 56

RAM Random Access Memory. 5, 7, 8, 49, 67, 86, 88

RF Radio Frequency. 3, 47, 49, 57, 105

RFID Radio Frequency IDentification. 59

RNG Random Number Generator. 1, 7

ROM Read Only Memory. 4, 5, 7, 8, 48, 49, 68, 86

RSA Rivest Shamir Adleman. 56, 74, 75, 92, *Nomenclature:* RSA

RTL Register Transfer Level. 13–15, 19, 20, 24

RX Reception. 37, 60, 76, 78, 80

SCLK Serial Clock. 31, 35

SCML SystemC Modeling Library. 67

SDL System Description Language. 21

SF SPI Framing Layer. 34

SFI Short File Identifier. 55

SHDLC Simple High-level Data Link Control. 34, 62, 77, 78

SIMD Single Instruction Multiple Data. 6

SISD Single Instruction Single Data. 6

SoC System-on-Chip. I, II, 12, 14, 19, 25, 26

SOF Start of Frame. 29, 31, 60

SPI Serial Peripheral Interface. 31, 34, 35, 38, 98

SS Slave Select. 31, 35

SW Software. I, II, 10, 13, 19, 24, 48, 49, 57, 65, 67, 73, 76, 78, 84, 85, 95–97

SWP Single Wire Protocol. 28, 30, 34, 38, 51, 59, 60, 62, 66, 71, 76, 78, 80, 84–86, 89–96, 98

T-BCA Transaction-based Bus Cycle Accurate. 15, 26

TAP Test Access Port. 35

TCK Test Clock Input. 35

TDI Test Data Input. 35

TDO Test Data Output. 35

TLM Transaction Level Modeling. 16, 17, 19, 22, 23, 26, 27, 49, 59, 60, 63, 66, 67, 71, 73, 76, 77, 80, 83, 84, 95, 97

TMS Test Mode Select. 35

TX Transmission. 37, 60, 76–78, 80

UICC Universal-Integrated-Circuit-Card. 4, 28, *Nomenclature:* UICC

UMTS Universal Mobile Telecommunications System. 25

UVM Universal Verification Methodology. 23

VHDL Very high-speed Hardware Description Language. 20

WDT Watch Dog Timer. 7

A.2 Nomenclature

AES The Advanced Encryption Standard is a symmetric-key block-cipher cryptosystem. This means, the same key is used for en- as well as decryption and the key-size specifies the block-length of the plain- as well as ciphertext. 74

bus In computer architectures, the bus is a HW-module which is responsible for data transfers inside or between computers. From the physical point of view, it is often simply a bundle of wires connecting multiple components with each other with an addtitional arbiter implementing the logic for address translation and handling simultaneous accesses by applying a priority-scheme or something similar. II, 10, 13–17, 25–27, 36, 38, 40–45, 49, 60, 62, 63, 65–67, 71, 73, 80, 81, 83, 86, 87, 89–98

CLF Contact-Less-Frontend - a HW unit responsible for contactless data transmission and reception via an electro-magnetic field. 28

co-processor A dedicated HW unit for a special purpose helping the CPU to decrease its workload. 1, 5, 7, 17

GDSII Graphic Data System II - a de-facto standard for layout-data of integrated circuits. 14

IEEE Institute of Electrical and Electronics Engineers - A professional association of technical professionals formed in 1963. See also `http://www.ieee.org/index.html`. 19, 22, 35

reader A smartcard reader is a device that communicates with the smartcard. This is done either physically via wires for contact-based smartcards or via the RF-field generated by the reader for contactless smartcards. 5, 28, 49, 53, 59, 86, 92, 95, 96

RSA Public key cryptosystem based on the integer factorization problem invented in 1977. RSA stands for the surnames of Ron Rivest, Adi Shamir and Leonard Adlemen who are the inventors. 56

UICC Universal-Integrated-Circuit-Card - a smart card in mobile devices used e.g. for identification in the telephone network. 4

Bibliography

[A$^+$01] IEEE Standards Association et al. *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE, 2001.

[ABR$^+$03] Manoj Ariyamparambath, Denis Bussaglia, Bernd Reinkemeier, Tim Kogel, and Torsten Kempf. A Highly Efficient Modeling Style for Heterogeneous Bus Architectures. In *System-on-Chip, 2003. Proceedings. International Symposium on*, pages 83–87. IEEE, 2003.

[an-11] SCSPI Protocol Specification - SPI for Smart Cards. March 2011.

[an1] AN11177 - Inter Processor Communication on LPC43xx. `http://www.lpcware.com/content/nxpfile/an11177-inter-processor-communication-lpc43xx`. Revision: 2012-03-19.

[Ayn09] John Aynsley. Osci TLM-2.0 Language Reference Manual. *Open SystemC Initiative (OSCI)*, page 15, 2009.

[BBBS07] E Barker, W Barker, WP Burr, and M Smid. NIST SP800-57 Recommendation for Key Management, Part 1: General. *Retrieved September*, 5(2009):800–57, 2007.

[byt] I^2C vs SPI. `http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/`. Accessed: 2014-08-22.

[CCL09] Lih-Yih Chiou, Yi-Siou Chen, and Chih-Hsien Lee. System-level Bus-based Communication Architecture Exploration using a Pseudoparallel Algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(8):1213–1223, 2009.

[CGO01] Lukai Cai, Daniel Gajski, and Mike Olivarez. Introduction of System Level Architecture Exploration using the SpecC Methodology. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 5, pages 9–12. IEEE, 2001.

[CVG03] Lukai Cai, Shireesh Verma, and Daniel D Gajski. Comparison of Specfic and SystemC languages for system design. *CECS, University of California, Irvine, CA, USA, Tech. Rep*, 2003.

[DGKO02] Rainer Dömer, Andreas Gerstlauer, Paul Kritzinger, and Mike Olivarez. The SpecC System-Level Design Language and Methodology. In *Parts1 & 2, Embedded Systems Conference*. Citeseer, 2002.

[dpc] DPCI: An Efficient Scalable System-on-chip Communication Architecture. `http://www.design-reuse.com/articles/15736/dpci-an-efficient-scalable-system-on-chip-communication-architecture.html`. Accessed: 2014-07-09.

[emv] EMV Contactless - The EMV Contactless Specifications for Payment Systems. `http://www.emvco.com/specifications.aspx?id=21`. Accessed: 2014-09-19.

[ETS08] ETSI. Smart Cards; UICC - Contactless Front-end (CLF) Interface; Part 1: Physical and data link layer characteristics (Release 7). Technical specification, ETSI, Sept 2008. `http://www.etsi.org/deliver/etsi_ts/102600_102699/102613/07.03.00_60/ts_102613v070300p.pdf`.

[Fly72] Michael Flynn. Some Computer Organizations and their Effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[G^+05] Frank Ghenassia et al. *Transaction-level Modeling with SystemC*. Springer, 2005.

[GB11] Mark Glasser and Janick Bergeron. TLM-2.0 in SystemVerilog. *Proceedings of the DVCON2011. Available at: events.dvcon.org/2011/proceedings/.../04_2.pdf*, 2011.

[GOC11] Marco Giammarini, Simone Orcioni, and Massimo Conti. Powersim: Power Estimation with SystemC. In *Solutions on Embedded Systems*, pages 285–300. Springer, 2011.

[GZD^+00] Daniel D Gajski, Jianwen Zhu, Rainer Domer, Andreas Gerstlauer, and Shuqing Zhao. SPECC: Specification Language and Methodology. 2000.

[Hea02] Steve Heath. *Embedded Systems Design*. Newnes, 2002.

[HML02] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with LISA*. Springer, 2002.

[I^+01] Open SystemC Initiative et al. SystemC Version 2.0 User's Guide, 2001.

[I^+06] Open SystemC Initiative et al. IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society*, 2006.

[Ini11] OSCI Open SystemC Initiative. IEEE 1666 Language Reference Manual, 2011.

[inta] An Introduction to the Intel QuickPath Interconnect. `http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf`. Accessed: 2014-07-09.

[intb] Introduction to Parallel Computing, EPCC. `http://www2.epcc.ed.ac.uk/computing/training/document_archive/decomp-course/Decomposing.book_5.html`. Accessed: 2014-05-23.

[jta] About Boundary-Scan. `http://www.jtag.nl/en/content/about-boundary-scan`. Accessed: 2014-08-22.

[Kub97] John David Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. PhD thesis, Citeseer, 1997.

[Lia04] Jian Liang. *Development and Verification of System-on-a-chip Communication Architecture*. PhD thesis, University of Massachusetts Amherst, 2004.

[Mar03] Peter Marwedel. *Embedded System Design*, volume 1. Springer, 2003.

[mas] MasterCard Contactless Performance Requirement - Application Note Number 7. `https://www.paypass.com/Card_TA/PayPassPerformMeasureAppNoteNumb7.pdf`. Accessed: 2014-10-15.

[mim] MIMD Basics. `http://ra.ziti.uni-heidelberg.de/pages/lectures/hws08/ra2/script_pdf/basics_mimd.pdf`. Accessed: 2014-05-23.

[nis] NIST - National Institute for Standards and Technic. `http://www.nist.gov/`. Accessed: 2014-10-03.

[PD10] Sudeep Pasricha and Nikil Dutt. *On-chip Communication Architectures: System on Chip Interconnect*. Morgan Kaufmann, 2010.

[Pöl12] Andreas Pöllabauer. Design und Implementierung einer Anwendungsanalyse für Smart Cards. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2012.

[PPM$^+$07] Vassilis Papaefstathiou, Dionisios Pnevmatikatos, Manolis Marazakis, Giorgos Kalokairinos, Aggelos Ioannou, Michael Papamichael, Stamatis Kavadias, Giorgos Mihelogiannakis, and Manolis Katevenis. Prototyping Efficient Interprocessor Communication Mechanisms. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pages 26–33. IEEE, 2007.

[RSG$^+$04] W Rosenstiel, S Swan, F Ghenassia, P Flake, and J Srouji. SystemC and SystemVerilog: Where do they fit? Where are they going? In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 122–127. IEEE, 2004.

[SD06] Gunar Schirner and Rainer Dömer. Quantitative Analysis of Transaction Level Models for the AMBA bus. In *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, pages 230–235. European Design and Automation Association, 2006.

[SPD08] Alena Simalatsar, Roberto Passerone, and Douglas Densmore. A Methodology for Architecture Exploration and Performance Analysis using System Level Design Languages and rapid Architecture Profiling. In *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, pages 95–102. IEEE, 2008.

[Vie14] Matthias Viertler. Evaluation of a SystemC Hierarchical-Bus Intersystem Communication Model. Seminar Work, Institute for Technical Informatics, Graz University of Technology, March 2014.

[WDL$^+$05] Andreas Wieferink, Malte Doerper, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tim Kogel, Gunnar Braun, and Achim Nohl. System Level Processor/Communication Co-exploration Methodology for Multiprocessor System-on-Chip Platforms. In *Computers and Digital Techniques, IEE Proceedings-*, volume 152, pages 3–11. IET, 2005.

[WR03] Wolfgang Effing Wolfgang Rankl. Smartcard Handbook, 2003.

[WWRL08] Zhonglei Wang, Thomas Wild, S Ruping, and Bernhard Lippmann. Benchmarking Domain Specific Processors: A Case Study of Evaluating a Smart Card Processor Design. In *Symposium on VLSI, 2008. ISVLSI'08. IEEE Computer Society Annual*, pages 16–21. IEEE, 2008.