Institute for Communication Technologies and Embedded Systems (ICE)
RWTH Aachen University
Prof. Dr.-Ing. Gerd Ascheid Prof. Dr. rer. nat. Rainer Leupers

Bachelor's Thesis: B-012

Design and Implementation of an SDHCI Model for Virtual Platforms

Entwurf und Implementierung eines SDHCI Modells für virtuelle Plattformen

by

Lasse Urban

Matr.-No. 369287

September 2019

Supervisors:
Prof. Dr. rer. nat. Rainer Leupers
Lukas Jünger, M.Sc.

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Urban, Lasse
Name, Vorname/Last Name, First Name

369287
Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit~~/Bachelorarbeit/ ~~Masterarbeit~~* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present ~~paper~~/Bachelor thesis/~~Master thesis~~* entitled

Design and Implementation of an SDHCI Model for Virtual Platforms

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 30.08.2019
Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen
*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 30.08.2019
Ort, Datum/City, Date

Unterschrift/Signature

Contents

1 Introduction

1.1 Motivation

Computers are at the heart of modern life. Everyone uses these machines, people complain when they work too slowly and are happy when a new generation of faster computers comes onto the market. But hardly anyone of the standard users knows how they really work. They know nothing about the interaction of hardware and software. They also do not know what an enormously important job the operating system does and they have never heard of bits and bytes being written to buses and registers.

Computer Engineering is the fascinating science that deals with these topics. When Konrad Zuse completed the world's first programmable computer, the Z3, in 1941 [1, p. 35], he could not imagine the importance and significance his field would have one day. From the day he introduced his Z3, engineers around the world began to develop and improve computers and the algorithms they execute. Von Neumann came up with his computer architecture model, the first semiconductor chips were developed, and over the years the integration density on these chips grew faster and faster. Moore's law - formulated in 1965 by Intel co-founder Dr. Gordon E. Moore - says that the packing density of transistors on a microprocessor and thus its performance in million instructions per second doubles about every 18 months [1, p. 40]. Nowadays, computers are no longer based on a single processor core, but more and more cores are integrated on a single chip. We have arrived in the so-called multi-core age. In this, also the integration of entire systems with many processor cores on a single chip is applied. This is called SoC - System on Chip.

However, operating systems have also become an increasingly important science. From one of the first operating systems, the MS-DOS 1.0 from Microsoft, which was widely used with IBM's "IBM PC" in 1981 [2], the development went very rapidly to today's Linux, Windows, IOS and Android operating systems. But we do not only encounter operating systems on computers, tablets and smartphones. Nowadays, every technical device has a control board that runs an embedded operating system. So, what is an operating system anyway? According to Herold [1, p. 423 f.], an operating system has two basic tasks: It is the interface between human and hardware and at the same time manages the resources of the computer system, e.g. processor(s), memory, and peripheral devices. So, it is obvious that a computer engineer inevitably has to be familiar with operating systems. They play a very important role in the development and integration of new hardware components because they contain the driver functions. The driver functions are the interfaces between the computer system, the operating system, and the new hardware devices. But let us get back to computer hardware first.

As shown, the complexity of computer hardware has increased dramatically over the last few decades. It is no longer so trivial to fully understand computer hardware and at the same time it is becoming more and more difficult to locate problems and bugs.
Another important point for computer engineers is the production cost of new hardware components as prototypes. When someone develops a new hardware component, it is perfectly obvious that he cannot build a new prototype in real hardware at every stage of development.

Nevertheless, he must test the behavior of the component and investigate and optimize the interaction between it and the host system.

For these reasons, a software-based simulation of hardware components - experts call it virtual prototyping - appears to be quite useful. From these virtual components one can then assemble an entire computer system. This is how the research area of Virtual Platforms (VPs) was born. VPs basically consist of a processor model, a memory model, an interrupt controller, a system bus model, and models of several peripherals - all implemented in software at a certain level of abstraction.

On this virtual computer hardware, a real Linux operating system can then be booted within a few seconds, i.e. quasi in real time. This Linux contains original Linux drivers to interact with the virtual peripheral components. Is it not fascinating and exciting that a real Linux with original drivers runs almost in real time on a virtual hardware?

This Bachelor thesis is about modeling such a hardware component. The task is to implement a fully functional virtual model of an SD Host Controller Interface (SDHCI). An SDHCI is a peripheral device that connects the host computer's bus system to one or any number of SD cards or SDIO devices. Figure 1.1 shows this intended function of the SDHCI model. The figure illustrates how the SDHCI controller connects an SD card to the system bus of the VP. It also shows the main components of a VP - the Central Processing Unit (CPU) and the memory model - introduced previously.

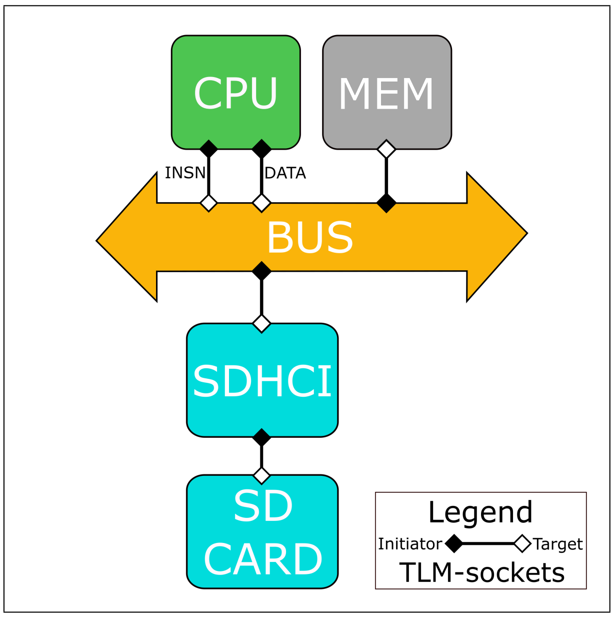


Figure 1.1: Overviev of the Virtual Platform and the intended SDHCI model

The SDHCI plays an important role in the VP. It can be addressed by the Linux operating system via the system bus, i.e. the Linux can send commands or data to it. The SDHCI controller has registers in which the data can be written. It then processes the commands and data and forwards them to the virtual SD card or writes the data to it. In the same way a read command can be initiated by the Linux. Then the SDHCI device reads the data from the SD card and makes it available to the operating system via its registers.

In the further course of the work, a Direct Memory Access (DMA) functionality is to be implemented, so that the SDHCI controller can write the read data into the main memory itself and the CPU no longer has to take it over. In the same way, the SDHCI controller should be able to read the data to be written from the main memory itself. Thus a speed-up is reached, since the CPU can be occupied then parallel with other tasks.

In this thesis the OpenRISC 1000 Multicore Virtual Platform (OR1kMVP) by Jan Henrik Weinstock [3] is used. The SDHCI model is integrated into this VP and tested with it. The model is published on GitHub [4] and can be also incorporated into any other VP based on SystemC TLM.

1.2 Related Work

Because VPs are widely used by large software companies nowadays, many have implemented solutions to communicate with virtual SD cards. However, their SDHCI models are not open source and can therefore not be used for research.

There are very few published works on SDHCI simulations. Nevertheless, the SDHCI model of the QEMU emulator is to be mentioned [5]. Two programmers emulated an SDHCI device according to the specification of the SD Assosiaction. However, this is not implemented in SystemC TLM, so the code is not suitable for use in VPs.

Imperas Open Virtual Platforms, on the other hand, contain three models of SDHCI devices [6]. They published the models NxpIMX6_SDHC, FreescaleVybridSDHC, and FreescaleKinetisSDHC. The registers implemented for these models are similar to those used in this Bachelor thesis. A difference between the models of the Imperas Open Virtual Platforms and the model developed in this thesis is the implementation of DMA. The models of the website have implemented Advanced DMA (ADMA), whereas in this thesis only Single Operation DMA (SDMA) is implemented. This is sufficient for most applications of VPs. Unfortunately, Imperas did not publish the source code their three SDHCI models.

2 Theoretical Foundations

2.1 Virtual Platforms

A Virtual Platform (VP) is an executable software model that can completely mirror the functionality of a hardware system [7, p. 17]. It is therefore a simulation of the complete computer hardware, which is assembled from many different virtual models of hardware components. These individual components are called virtual prototypes. A concrete example of a VP and from which virtual components it is composed can be found in section 2.5 on page 11. When implementing a VP of course a certain level of abstraction must be found. For the functionality of a certain component and the entire system it is not necessary to model the hardware down to the smallest detail. Since only the functionality and not the hardware details have to be implemented, the development of a VP requires much less development time than real hardware.

Normally, large software companies start the development of VPs parallel to the development of real hardware. For the reasons described above, the VP is finished much earlier than the real hardware. De Schutter mentions a time difference of 9 to 12 months [7, p. 21]. This time advantage can be used by software developers to start developing, debugging and validating system software earlier. Software development can be the development of boot code for an operating system (OS), drivers for peripheral devices, or specific software applications that are optimized for the architecture. When writing code, it must in most cases be debugged, i.e. executed step-by-step and monitored at specific breakpoints. This allows to quickly find and fix most flaws in the code. When debugging, the VP has three distinct advantages over real hardware.
First, visibility should be mentioned. With the VP, the system can be inspected with any degree of precision. This means that it can be looked into every simulated hardware block and the status of every single register and signal can be checked. This is not that easy with real hardware, because e.g. a small chip cannot be opened to measure and check its internal signals. At the same time, it is always clear at what point of execution the virtual hardware is and on what point the software - running on it - is. Parallel debugging is discussed more detailed in section 3.2 on page 18.
The second decisive advantage is control over the system. The software developer can stop the VP at any point and e.g. inspect and modify the memory. After that, the simulation can be continued from this point without much trouble. The uncomplicated stopping of subsystems or possibly the entire system at any point is not possible with real hardware.
The third and last advantage of debugging virtual hardware is that the simulation is deterministic. Deterministic means that at each point of execution it is unambiguous what the simulation will do next. Thus, an error can be exactly reproduced several times and examined under different aspects. With real hardware, an error can often not be reproduced exactly, because there are many external influences and time-critical parallelism.

The third essential component in the development of system software is, in addition to implementation (development) and debugging, validation and verification. This means that the software developer has to do countless tests. The written software as well as the hardware and their interaction have to be thoroughly examined. And here again the use of VPs has some advantages. The developer can implement tests in software that are executed on the host system and test the behavior of the developed software and the (virtual) hardware in parallel, similar to debugging. Nowadays, this is extensively used in large software companies. They acquire extra computer infrastructure that is only used to test the software, developed during the day, at night. Such unit tests and test-driven development will be discussed more in detail in section 3.3 on page 19.

In summary, it can be said that VPs are used in many industrial companies engaged in software development. They use them to develop and debug software in parallel with the hardware it is running on and to validate the behavior of the overall system. This shortens the development time, improves the quality of the product and thus maximizes profit. [7]

2.2 SystemC

The de facto standard for the implementation of virtual prototypes and thus VPs is SystemC TLM-2.0. TLM stands for Transaction-Level Modeling. It models the transaction of commands and data between an initiator (sender) and a target (receiver). TLM provides software blocks from which virtual components can be built. The blocks are mainly used to connect the components easily and to enable a simple communication between them. TLM models are implemented in the hardware modeling language SystemC, so this section is about SystemC. TLM-2.0 in detail will be discussed in the next section 2.3.

Actually, SystemC is only a class library for C/C++ and not a programming language like e.g. VHDL. However, because it has its own IEEE standard (IEEE Std. 1666-2011) [8], which defines it and holds more than 600 pages, it is often called a programming language. Here is a short overview of the SystemC simulation kernel and the most important modules and functions like `sc_module`, `sc_thread` and `sc_method`. For a deeper understanding of the next section and the implementation part in chapters 4 and 5, the reader is recommended to study the SystemC Standard [8].

A SystemC application is an event-based, discrete real-time simulation consisting of two phases. The first phase that the SystemC simulation kernel performs is the so-called Elaboration phase. As Figure 2.1 shows, in this phase the kernel builds up a module hierarchy by instantiating and connecting all modules. This is comparable to an electronic construction kit with a breadboard, only in a software simulation. At the beginning, the individual components, e.g. ICs, are taken out of the kit and plugged onto the breadboard. These are the so-called sc_modules. They are instantiated by calling the constructor `sc_module()`. The pins of the ICs that are inserted into the breadboard correspond to the so-called sc_ports. These are the connections of the module to the outside. There are three important port types: sc_in, sc_out and sc_inout, which have a data type that they can transfer. This is specified in '<>' after the port type. Besides the sc_ports there are sc_exports which are not interfaces to the outside but to the parent modules. Furthermore, the so-called primitive channels exist, which are bound to the ports. These include, for example, sc_signals, which also have a data type. The sc_signals correspond to the wires in the example of the breadboard. From these basic components a hierarchy of modules is built up in the Elaboration phase. The modules are connected via sc_ports and channels and can exchange values with each other. In addition, processes can be started in the Elaboration phase, which will be described later.

After the Elaboration phase, the Simulation phase is started. Figure 2.1 shows the five steps executed in the Simulation phase. The first one is the initialization of the simulation. Thereby, the Update phase, which is described below, is performed once until the Delta Notification

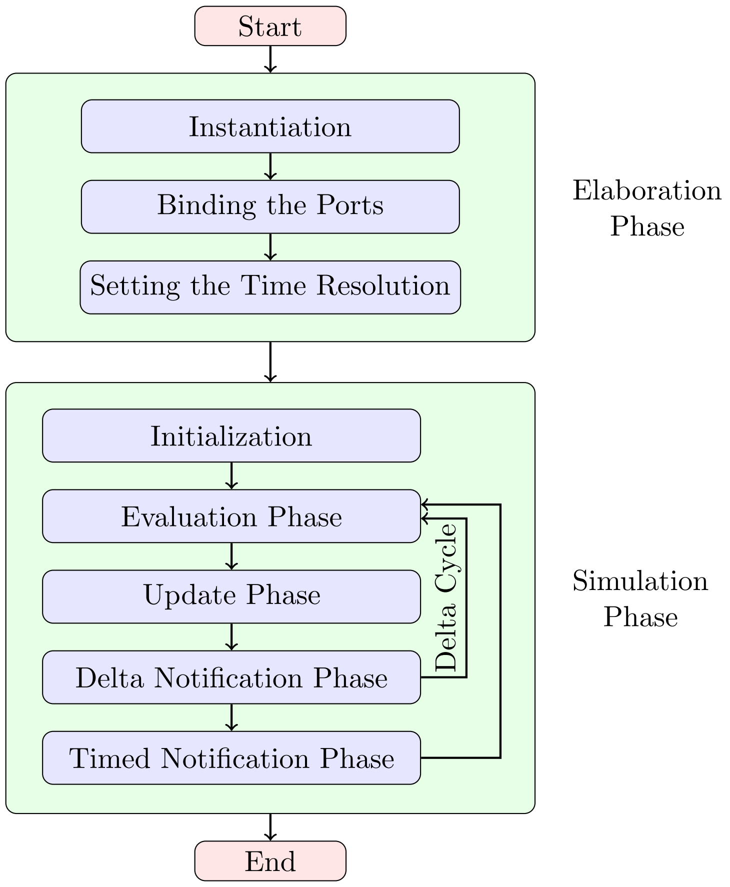


Figure 2.1: Phases of a SystemC simulation

phase. Then, all instantiated processes are added to the pool of runnable processes. Afterwards, a Delta Notification phase is executed. After the Initialization phase, the phases are performed as shown in Figure 2.1. First, an Evaluation phase is performed in which the current values of all signals, ports, etc. are calculated. These are then updated in the Update phase. Since SystemC is event-based and sensitivity dependencies can be defined, a delta cycle is made after this. The most important thing about a delta cycle is that the simulated time does not advance. Besides, not all signals, ports etc. have the correct values at this point. As the loop in Figure 2.1 shows, the simulation kernel then returns to the Evaluation phase. It checks whether dependent values changed due to the mentioned events or sensitivities. Sensitivities will be explained when sc_threads and sc_methods are introduced. The values are updated in the following Update phase if necessary. This procedure is done until no values change in the current time step due to any dependencies. Then a real time step is made, i.e. the simulation time jumps forward to the point when the next event from the event queue must be handled.
The whole simulation runs until the simulation time has expired or the simulation is manually stopped by the user by calling `sc_stop()`. [8, p. 12ff]

One of the most important features and benefits of SystemC is that parallelism can be simulated. Therefore, it is important to know that there are processes in SystemC. A process is an execution line of instructions and several processes can run concurrently. However, there are three kinds of processes: sc_thread, sc_cthread and sc_method. If an sc_module contains one or more processes, the macro `SC_HAS_PROCESS(...)` must be inside the module. Subsequently, the process can be defined with a process macro inside the constructor of the module, e.g. `SC_THREAD(run)`. In the brackets a function is specified, which is the process, in this example the function `void run()`.

Furthermore, the differences between the process types should be pointed out. An sc_thread is executed exactly once and then it terminates. Therefore, endless loops are often used in a thread. Sc_cthreads are clocked threads, i.e. there is a clock specified that triggers it. Sc_methods are processes that keep on running, i.e. they are executed repeatedly and not just once. Processes can be triggered e.g. by sc_events, clock events (sc_cthreads), or other events if they got a sensitivity list in the constructor of the module. At this point the concept of sensitivity in SystemC should be introduced. In the constructor of module, where processes, signals and ports are defined, sc_threads and sc_methods can be defined as sensitive to any events, e.g. a function call or a rising edge of a port value. This means that every time this event occurs the simulation kernel calls the thread or method that is sensitive to this event. Sensitivities can be defined after the definition of the threads or methods with the sensitivity operator: `sensitive << ` *`event`* `;`.
Moreover, sc_threads can be paused by calling `wait()`. Then they are continued when an event occurs they are statically sensitive to. If `wait()` gets the parameter SC_ZERO_TIME, the process is continued in the next delta cycle. Alternatively, it can be waited for a specific event. This also allows process synchronization.

The last two features of processes worth mentioning are the functions `dont_initialize()` and `sc_spawn()`. When calling the function `dont_initialize()` in the constructor of the module, it can be ensured that the process is not included in the pool of runnable processes during the Initialization phase. This has the advantage that processes like this can only run if they are triggered. With the function `sc_spawn()` processes can be created dynamically. Normally, processes are created statically in the constructor and run or wait from the beginning. In contrast, `sc_spawn()` creates a process at runtime that is directly runnable.

At this point, it should be pointed out once again that the description of SystemC given here reflects only a small part of its possibilities. Actually, this does not do justice to the power of SystemC. The reader is once again referred to the comprehensive SystemC standard [8].

2.3 Transaction Level Modeling (TLM-2.0)

As introduced in the previous section, TLM stands for Transaction Level Modeling. The current standard is TLM-2.0 and is an extension of the SystemC standard. This makes sense because the TLM models are built from SystemC elements and the SystemC simulation kernel is used. Therefore, the TLM standard can be found in [8] beginning on page 413.

In general, TLM-2.0 consists of a set of core interfaces, the global quantum, initiator and target sockets, the generic payload and base protocol, and some utilities. Each of these five main components is briefly explained in the following.
However, before the main elements of TLM can be explained, TLM coding styles must be discussed. There are three different coding styles in TLM: untimed, loosely-timed and approximately-timed. The untimed coding style means that no consideration is given to the simulation time. This is allowed in SystemC, because synchronization in the transmission of data can also be achieved explicitly via waits or semaphores. Semaphores are exclusion or blocking variables. In TLM-2.0 untimed modeling is not intended, therefore it will not be explained here more in detail.
Loosely-timed coding style, on the other hand, can use the simulation time for synchronization. In the case of a so-called blocking_transport, which is always associated with the loosly-timed style, there are exactly two time marks. The first mark is the beginning of the transport request, i.e. the call of the function `b_transport()`. The second mark, to which both the sender and the receiver must adhere, is the beginning of the response. These two

time stamps are sufficient to guarantee a reliable, more or less time-accurate communication without the need for explicit synchronization with each transmission. Thus the loosely-timed coding style is the optimal choice for the implementation of VPs [8, p. 416f]. It allows the modeling of bus systems, timers and interrupts and also is accurate enough to boot an OS on the VP. At the same time, it only requires a minimal number of synchronization points, so that the performance of the VP is not noticeably weakened by the TLM communication. In addition, the loosely-timed coding style supports temporal decoupling. This means that certain processes may run up to a certain point in time - a global quantum or until a value depends on another event - ahead of the simulation time. This again can increase the performance.

The third and last coding style is the approximately-timed coding style. It is more accurate in time than the loosely-timed style but also requires four time marks. These indicate the beginning and the end of the two phases request and response of the communication. Due to the demand for a higher time accuracy, no temporal decoupling is possible here. This coding style is suitable for architecture exploration and performance analysis.

Thus, the loosely-timed coding style is the optimal style for the implementation of VPs, which is what this work is about. Therefore, the explanation of the five main components of TLM-2.0 focuses on this coding style.

The first important component of TLM are the core interfaces. TLM-1 already had unidirectional and bidirectional (transport) interfaces. TLM-2.0 then added the blocking and non-blocking transport interfaces as well as the direct memory interfaces (DMI) and the debug transport interfaces. As already mentioned above, the blocking transport interfaces play the most important role in the loosely-timed coding style. They possess the pure virtual function `b_transport()`, which performs the transaction between two modules. The function requires two arguments. The first one is a non-constant reference to a TLM generic payload, which will be introduced below. The second argument is a time value indicating the start and end of the transaction.

Secondly, TLM-2.0 provides a global quantum. This is an sc_time, which is defined globally for all transactions in the header `tlm.h`. The advantage of this is that all transactions use the same time quantum and that e.g. in the case of temporal decoupling a fixed limit is defined up to which a process may run ahead of the simulation.

Thirdly, the sockets must be mentioned. There are tlm_initiator_sockets, which initiate transactions, and tlm_target_sockets, which receive them. These sockets have sc_ports of the type of the interfaces defined above. Figure 2.2 illustrates how tlm_initiator_sockets and tlm_target_sockets are connected via the interfaces.

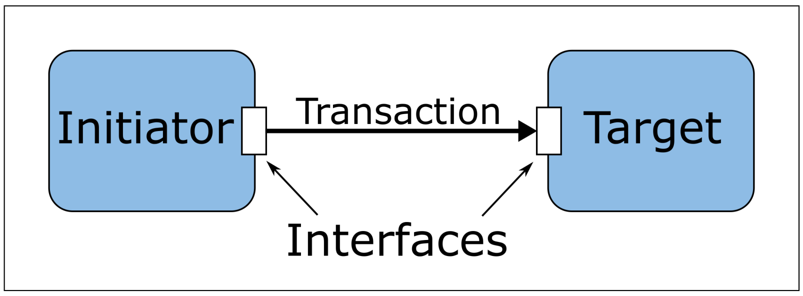


Figure 2.2: TLM Transaction between a TLM Initiator Socket and a Target Socket

Via these interface ports, data packets can be sent which have already been introduced above as tlm_generic_payload. A generic payload typically contains attributes such as a tlm_command, an address, data, and a response status. The generic payload is closely linked to the TLM base protocol, in which rules for communication are defined to ensure the best and smoothest possible interoperability between different TLM models. In addition to rules

for the transmission of generic payloads, the protocol also defines the behavior of tlm_sockets, the described transport interfaces, and the transmission phases request and response.
The last main component of the TLM standard are the utilities. Utilities are a set of classes that contribute to convenience for the programmer and consistency of coding style. As an example, the convenience sockets should be mentioned. Convenience sockets include, for example, the simple_initiator_socket and the simple_target_socket. These are derived from the original sockets and provide, among other things, the possibility to simply register callback functions.

With this knowledge about SystemC and the main features of TLM-2.0, the reader should now be able to understand the concrete implementation of the virtual prototypes and the entire VP.

2.4 Virtual Components Modeling Library (VCML)

Now that the basics of SystemC and TLM are known, it is time to move on to the practical side. First, the Virtual Components Modeling Library (VCML), which is used by the simulation team of the Chair for Software for Systems on Silicon at RWTH Aachen University, will be introduced. It was implemented by Jan Henrik Weinstock and is published on GitHub [9].

VCML has two basic goals. On the one hand, it aims to simplify the use of TLM models and protocols explained in the previous section. This will make it easier to implement new models and components for VPs. With this goal, first of all, the introduction of vcml::master_sockets and vcml::slave_sockets should be mentioned. These are derived from the simple_initialor_socket and simple_target_socket described in section 2.3 and add e.g. simpler access, read, and write functions. In addition, VCML provides a basic skeleton for virtual hardware components of the VP. If a virtual hardware component is to be implemented, it can simply be derived from the class vcml::component. By adding vcml::slave_sockets and vcml::master_sockets to the component instance, a virtual hardware component is created that is able to send and receive TLM transactions. To initiate or respond to transactions, the predefined functions of the class vcml::component, e.g. `b_transport`, can be used or overwritten. This is exactly what was done in the case of the class vcml::peripheral. It is derived from the class vcml::component and provides the basic skeleton for the implementation of virtual Peripheral Input/Output (PIO) devices. It adds, for example, a number of convenience functions such as read and write callback functions. Furthermore, it introduces a vector that holds virtual hardware registers. These can be instantiated from the class vcml::register.
VCML also contains the basic framework for Serial Peripheral Interface (SPI) communication, SD functions, processors, and lots of useful auxiliary classes. In the auxiliary classes e.g. memory ranges, debugging functions, logging functions, and the important vcml::properties are defined. The latter wraps variables and model properties so that they can be passed on and modified in a standardized way. Moreover, the vcml::properties can be changed at runtime via the command-line.
The second basic goal of VCML is to provide whole models of hardware components that are ready to use. From this pool of models, the programmer can assemble a VP. The hardware models can be found in [9] under `include/vcml/models` and `src/vcml/models`. They subdivide into models of ARM components, OpenCore components, and generic components. The latter are generic because they are not directly modeled after real hardware. These include, for example, the system bus model and the model of an SD card, which are particularly important for the implementation of a VP. VCML also contains the special simdev device. It is a generic device and was introduced to allow the manipulation of VP from "the inside". Among other things it contains a *Stop* register. When this register is written, the simulation is stopped by calling `sc_stop()`.

The model of the SDHCI [4], which is developed in the context of this Bachelor thesis, is an addition to the VCML and also belongs to the generic models.

2.5 OpenRISC 1000 Multicore Virtual Platform (OR1kMVP)

The OpenRISC 1000 Multicore Virtual Platform (OR1kMVP) [3] is a VP, which was developed by Jan Henrik Weinstock. It is OpenRISC based, i.e. Jan's GitHub repository contains a virtual model of an OpenRISC processor. This can be found in `src/or1kmvp/openrisc.cpp` [3]. The processor model is derived from the class processor of the VCML and also inherits from the class or1kiss::env. The OpenRISC 1000 Instruction Set Simulator (OR1kISS) is the instruction set simulator used by the OR1kMVP, which Jan also implemented himself [10]. An instruction set simulator is a simulator that mimics the behavior of a real processor. It contains processor registers and internal variables, e.g. a program counter. The processor model `openrisc.cpp` has inherited an OR1kISS environment or1kiss::env and contains a private pointer to an OR1kISS instance. In addition, it provides functions for connecting the processor to GDB and contains the interrupt ports for connecting peripheral devices. For more information on GDB see section 3.2 on page 18.

The described processor model is then used as a CPU to build the OR1kMVP. This is assembled in `src/or1kmvp/system.cpp` [3] from one or more processors and the devices of the VCML. The number of CPUs is specified in a config file loaded when starting the VP, e.g. the up.cfg in the `config` directory [3]. In the up.cfg the variable system.nrcpu is set to 1. The global time quantum and the device tree blob to be loaded are also specified in the config file.

As indicated, the OR1kMVP consists of the OpenRISC processor and the virtual components implemented in the VCML. Figure 2.3 shows how the result of the instantiation in `src/or1kmvp/system.cpp` [3] looks like. In addition to the processor, the OR1kMVP has an instance of the memory model and the system bus model from VCML. Many peripheral devices are attached to this system bus. Furthermore, some peripherals must be able to send an interrupt to the CPU. For this purpose the interrupt output ports (IRQ ports) of these devices are connected to the interrupt ports of the processor. This is shown by the dashed lines.

Note in particular that the OR1kMVP has an SPI controller which is an interface between the system bus and the SPI bus. To this SPI bus a generic device "spi2sd" is connected which converts the SPI commands into SD commands so that the SD card in the OR1kMVP is addressed via the SPI bus. This is shown in blue. In the context of this Bachelor thesis, the OR1kMVP is extended with an SDHCI device and an additional SD card. This is represented in red in Figure 2.3. The SD card and its connection to the system is important for the OR1kMVP, because it contains the Linux file system explained in section 2.6.

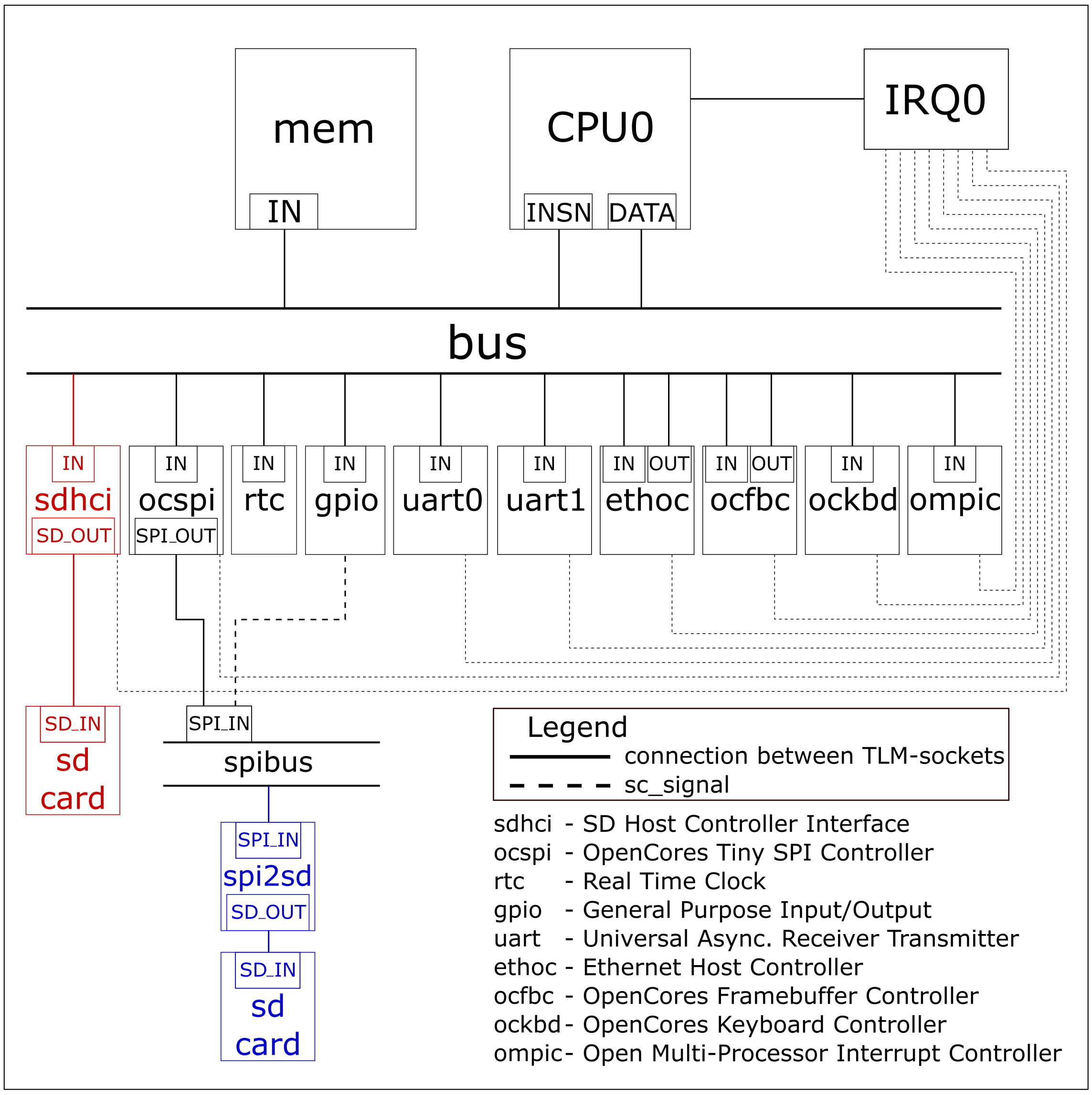

mem
IN
CPU0
INSN
DATA
IRQ0
bus
IN
sdhci
SD_OUT
IN
ocspi
SPI_OUT
IN
rtc
IN
gpio
IN
uart0
IN
uart1
IN
OUT
ethoc
IN
OUT
ocfbc
IN
ockbd
IN
ompic
SD_IN
sd
card
SPI_IN
spibus
SPI_IN
spi2sd
SD_OUT
SD_IN
sd
card
Legend
connection between TLM-sockets
sc_signal
sdhci - SD Host Controller Interface
ocspi - OpenCores Tiny SPI Controller
rtc - Real Time Clock
gpio - General Purpose Input/Output
uart - Universal Async. Receiver Transmitter
ethoc - Ethernet Host Controller
ocfbc - OpenCores Framebuffer Controller
ockbd - OpenCores Keyboard Controller
ompic - Open Multi-Processor Interrupt Controller

Figure 2.3: Bindings of the OpenRISC 1000 Multicore Virtual Platform

2.6 Operating System - Linux Kernel

Linux generally denotes a family of unix-like operating systems. This includes, for example, *Ubuntu*, *Debian*, or *CentOS*. All these operating systems (OSs) are based on the same kernel, the Linux kernel. It was published by Linus Torvalds in 1991 and is free and open-source. Therefore, the source tree for this kernel can be found on GitHub [11].

As explained in section 2.1, VPs simulate the hardware of computers. On these VPs an operating system is to run, in most cases it is a Linux OS. Exactly the same applies in the case of the OR1kMVP implemented by Jan Henrik Weinstock. His VP can in fact run a complete operating system, thus it can boot the Linux kernel. After the boot, a shell is opened where the user can enter instructions. To boot the kernel, a root file system and a device tree are needed. The latter is needed to make Linux know all hardware devices. The shell then provides access to the file system and the devices.
In the following, it will be described how the Linux kernel can be configured and built for the application in the OR1kMVP. Additionally, this section explains what the device tree exactly does and where the file system is located in the OR1kMVP.

In order to boot a kernel, it must be in `.elf` format. The file extension `.elf` stands for *Executable and Linking Format* and is the standard binary format for executable programs. The required `.elf` file of the Linux kernel can be built from the kernel source code from GitHub [11]. The GitHub repository must be cloned and then the command `make -j4 ARCH=openrisc CROSS_COMPILE=or1k-elf- O=BUILD all` must be executed in a console within the cloned repository. This will crosscompile the kernel. Crosscompiling means that the kernel is built for a particular computer architecture - in this case an OpenRISC architecture - on a foreign architecture. After the crosscompilation, the built file can be found in the `BUILD` directory. If changes are to be made to the kernel, this can be done in the configuration menu before building. This menu can be accessed with the command `make -j4 ARCH=openrisc CROSS_COMPILE=or1k-elf- O=BUILD menuconfig`. The extension of the Linux kernel via the configuration menu is used and explained in section 4.2 on page 22.
After the Linux kernel was built as an `.elf` file, it is an executable binary, but the OR1kMVP still can not boot it because it needs raw binary. To convert the `.elf` file to raw binary, the command `objcopy` can be used. It copies the executable binary and discards all redundant information. For example, the `objcopy` command of the or1k module can be used: `or1k-elf-objcopy -O binary BUILD/vmlinux ../or1kmvp/sw/vmlinux-4.20.0`. This will copy the binary code directly to the software folder of the OR1kMVP. In this form the OR1kMVP can boot the kernel. Additionally, the `.elf` file should be copied into the software folder of the OR1kMVP. This is necessary to debug the kernel with GDB. Debugging with GDB is explained in section 3.2 on page 18. The GDB debugger needs the information again that is discarded by the `objcopy` command.

In order to boot the built Linux kernel, there must be a device tree that is specific for the hardware on which the kernel is to be booted. The device tree contains all necessary information about the system and all hardware components [12]. It has a tree-like structure, i.e. it consists of individual nodes, and has three main components. First of all, there is the compatible string: `compatible = "openrisc, or1kmvp"`. It defines the system name and the developer. Secondly, the device tree contains a node `cpus`, in which all CPUs of the system are specified. Finally, for each additional hardware device there is a device tree entry. In section 4.1 on page 21 the device tree entry for the SDHCI controller by Fujitsu is shown and explained.

The device tree is usually available as device tree source (`.dts` file). In order to be readable by the Linux kernel, it must first be compiled just like the kernel. To do this the device tree compiler dtc can be used. It converts the `.dts` file into a `.dbt` file. This is a device tree blob and readable by the Linux kernel. The use of the dtc to compile the device tree source is described in detail in section 4.1 on page 22.

Hence, if there is a compiled and an 'objcopied' Linux kernel and a compiled device tree, the kernel can be booted on the OR1kMVP. Once the CPU is configured and running, the Linux kernel can deal with the peripheral hardware devices. It walks through the available drivers for peripheral devices and checks in the device tree blob whether there is suitable hardware available. This assignment of drivers and hardware devices in the device tree is done by a string comparison. Both the driver and the device tree entry have got a compatible string, which has to match in order to use the driver for the device. How this is done in the special example of the SDHCI model is described in section 4.5.2 on page 27.

At the end of the boot process the above mentioned file system is used. This is the directory structure, so to speak, with which the user can then work via the shell. The most commonly used file system with Linux is the ext4 (fourth extended file system). It is stored on the SD card of the OR1kMVP and has therefore to be loaded from the SD card at the end of the boot. This is why fast communication with the SD card is so important for the OR1kMVP. For this reason an SDHCI model is to be implemented in the context of this Bachelor thesis, which is supposed to accelerate the communication. Where exactly the kernel finds the file system on the SD card is indicated in the device tree. Therefore, the entry *root=/dev/mmcblk0p1* is added to the boot arguments.

2.7 SD Host Controller Interface (SDHCI) and SD Cards

An SD Host Controller Interface (SDHCI) is a hardware device that acts as an interface between a computer system and an SD card. It is often referred to as an SD host controller (SDHC), but because this can easily be confused with the High Capacity SD card - also abbreviated as SDHC - the term SDHCI is used throughout this work. The behavior and the functions that an SDHCI must have are defined in the "SD Host Controller Simplified Specification" [13]. This was written by the SD Association. The SD Association is an association of many mainly American technology companies that agreed on a uniform standard for the so-called SD cards in January 2000. The SD card was derived from the Multi Media Card (MMC) and is backward compatible to this technology until today. Initially, only the normal SD memory card was released, which every user of computers and smartphones should know until today. However, the SD card family was quickly expanded, as the SD Association recognized the potential of the SD interface and protocol. Besides the standard SD memory card and the mini- and micro-SD cards, the SD Input/Output cards (SDIO) were developed, which do not focus on the memory but on the SD interface. [14]
The family of SDIO devices includes for example Bluetooth SD cards, GPS SD cards, and Camera SD cards. The advantage is that via the SD card interface all these cards can be accessed very easily. In 2013, the SD Association introduced the so-called Intelligent SDIO (iSDIO). This denotes Combo SD cards, which combine the SDIO functionality with a large memory. For example there are Wireless Lan SD cards or Transfer Jet SD cards. Transfer Jet is a new type of short-range communication link. As described, the SD card interface has many interesting features and advantages, but most people are not aware of them. This is probably caused by the fact that the SD card interface has always been in competition with the Universal Serial Bus (USB) family. Nevertheless, the SD interface and the SD protocol are often used in low-priced SoCs. In other words, SD card technology also plays a role in embedded systems.

However, the SDHCI as a hardware device works as an interface between the system bus and any kind of SD cards, whether they are SD memory cards, SDIO cards, or iSDIO combo cards. Since in the application with VPs mostly only the standard SD memory cards are used, the implementation of the SDHCI model in the context of this work will focus on the communication with normal SD memory cards. The SDIO functionality can then be added later.

2.8 Direct Memory Access (DMA)

Direct Memory Access (DMA) is a hardware mechanism that enables a peripheral device to write data to the main memory or read data from it. The important advantage of this is that the CPU no longer is occupied with the data transfer, but can dedicate itself to other tasks.

Normally, a peripehrial device works as follows: The CPU passes a set of data and a command to the device, waits until it has completed its task, and copies the result data back into the main memory. The problem is that the CPU has to wait for the device and has to send all data to the device itself. This is connected with a high administration effort for the CPU and has therefore a high time-expenditure.
For this reason, DMA was invented, which gives the peripheral device access to the main memory. The CPU reserves a large memory area in the main memory as a DMA buffer and copies data into it if necessary. Then, it passes the address to the peripheral indicating the beginning of the DMA buffer in the main memory. This can be done for example via a *DMA Address* register. In addition, the device is usually given a boundary indicating how large the DMA buffer is. The peripheral device must then retrieve the data from the DMA buffer in the main memory itself. Afterwards, it can work with the data and do all the assigned tasks. When all tasks are finished, the peripheral device has to write the result data requested by the CPU back into the DMA buffer. If the DMA boundary is exceeded, the device can of course request a new DMA address from the CPU in the meantime. Otherwise, it informs the CPU only when all requested data is written into the main memory. This can be done for example with a simple **DMA** Interrupt. So it is visible that the CPU has nothing to do with the data transfer anymore when using DMA. All the work has to be done by the peripheral, while the CPU can do other tasks in parallel. Therefore, the introduction of DMA to a system normally brings a significant increase in data throughput and thus in performance.
However, it has to be considered that DMA is very close to hardware and therefore, it has to be implemented hardware specific. Thus, it is difficult to program uniform drivers for DMA data transport, but nowadays this is being worked on. [15]

An SDHCI supports two different types of DMA that are described in the "SD Host Controller Simplified Specification" [13]. These are Single Operation DMA (SDMA) and Advanced DMA (ADMA). SDMA is the basic form of DMA and allows to execute exactly one SD command per DMA transfer. As described above, when using SDMA, a new DMA address must be requested from the CPU when the buffer boundary exceeds. ADMA prevents this by allowing multiple DMA addresses to be specified in advance in a descriptor table. The DMA transfer can therefore be better pre-planned. This has the advantage that when transferring large amounts of data, the CPU is not constantly interrupted in its work to provide a new DMA address. The "SD Host Controller Simplified Specification" [13] even defines two types of ADMA, namely ADMA2 and ADMA3. ADMA2 allows only one ADMA transfer at a time, while ADMA3 allows multiple ADMA read and write transfers to be programmed in the descriptor table. This is suitable for transferring very large amounts of data.

Whether an SDHCI Controller supports SDMA or a kind of ADMA, is displayed in the *Capabilities* register (see section 4.4). Through this register, the SDHCI driver knows what kind of data transfer it can perform with the controller.

3 Working Methods

After the introduction of all theoretical basics, now the question arises how to approach the development of a SD Host Controller Interface (SDHCI) model for Virtual Platforms (VPs). Should the whole model be implemented first and then incorporated into the simulation to test it? And where to start looking for errors? How can one debug and test a VP? These questions will be answered in this chapter. First, the general working approach, then the debugging and finally the testing of the SDHCI model to be implemented will be explained.

3.1 General Approach

The development of a virtual prototype of a hardware component actually has a relatively uncomplicated approach. First, the basic framework of the virtual component has to be developed. This means that a class should be created for the component and this should ideally be derived from the vcml::component class or the vcml::peripheral class. For a more detailed explanation of these classes, see section 2.4 on page 10. Thereafter, an instance of this class, i.e. of the virtual model of the hardware component, should be created and incorporated into the VP. This is illustrated in section 4.3 on page 23 with a concrete example. Afterwards, the procedure shown in figure 3.1 should be applied iteratively.

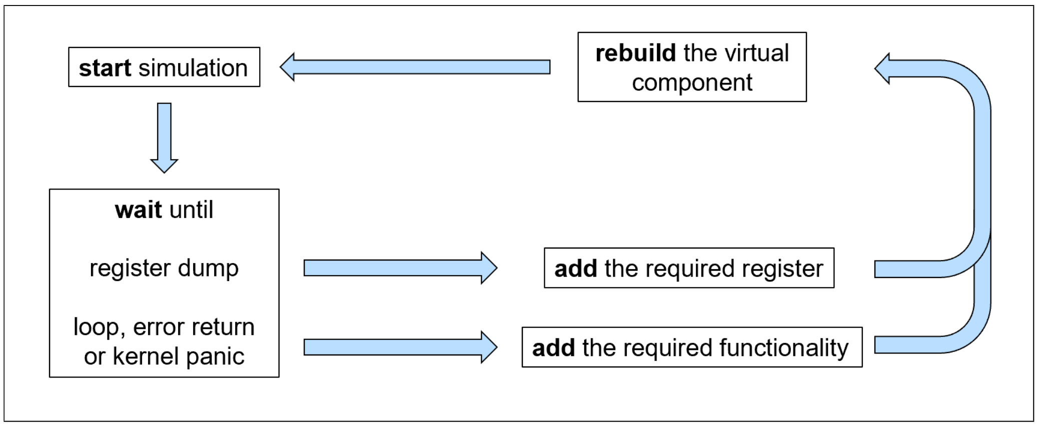

start simulation
rebuild the virtual component
wait until
register dump
loop, error return or kernel panic
add the required register
add the required functionality

Figure 3.1: General working approach

The figure shows that the simulation of the hardware, i.e. the VP, is simply started. Then, the programmer should wait for what happens. Since the new model to be implemented has neither the appropriate interfaces nor the desired functionalities, the VP will crash at the first access to the new model and the attempt to initialize it. Then, it has to be investigated why the VP crashed. This can be done, for example, by debugging the virtual hardware and debugging the Linux kernel. Both are explained in the next section 3.2. During the development of the SDHCI model that was implemented as part of this work, two types of errors often occurred. On the one hand, there can be a failure in the simulated hardware of

the SDHCI component. This may be the case, for example, if a register to be accessed has not been implemented yet. Then, as shown in figure 3.1, a *register dump* occurs. This type of error is usually very easy to recognize and fix because the *register dump* error message always indicates what caused the error.
The second type of errors that can occur are errors that affect the logical functionality of the model. These are usually not that easy to locate and fix because they often affect the interaction between the virtual hardware and the Linux kernel. As indicated in Figure 3.1, these can be errors that can result in endless loops, Linux kernel error messages, or even a kernel panic. These results can happen, for example, when the kernel expects an action from the model but it does not perform the required action or performs it incorrectly. This can be the case if the internal functionality of the model has not yet been implemented or has been implemented incorrectly. Then, by debugging the kernel and the virtual hardware in parallel, it must be determined what the kernel expects and what the hardware does. This parallel debugging is also described in section 3.2. After the bug is found, it needs to be fixed, although fixing it is usually easier than finding it.

As shown in figure 3.1, the model needs to be recompiled after finding and fixing the bugs. The simulation can then be restarted. Normally, it should now crash at a later point, i.e. there should be an increment in the simulation. The whole procedure shown in the figure is then performed iteratively, i.e. it is performed until the new virtual component has all necessary functions and the boot of the Linux kernel on the VP is working again. In addition, all functions, which are not needed during boot, should be tested with unit tests. This is described in section 3.3 on page 19.

3.2 Debugging the Virtual Hardware and the Linux Kernel

A debugger is a programming tool that can be used to search for errors in programs by going through it step by step [1, p. 142]. As indicated in the previous section, two debuggers are needed to debug a VP. One to debug the Linux kernel and the module driver and one to debug the virtual hardware, which is implemented in SystemC TLM. To Debug the Linux kernel, the GNU debugger (GDB) is used. In the following, this will be explained first. After that it will be explained how to debug the virtual hardware of the VP. To do this the debugger of the Eclipse IDE is used. Figure 3.2 illustrates the use of these two debuggers for parallel debugging of the kernel and the virtual hardware.

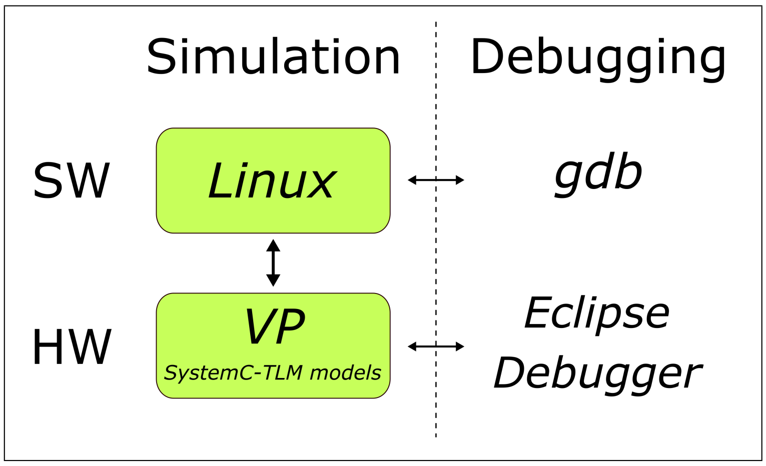


Figure 3.2: Parallel debugging of the Linux kernel and virtual hardware

As mentioned in section 2.5, the OpenRISC processor of the OR1kMVP, which runs the Linux kernel, has some useful functions to be connected with the GDB debugger. When starting the simulation of the VP in a Linux terminal with *bin/or1kmvp -f config/up.cfg* the vcml::property system.cpu0.gdb_wait can be changed, which tells the simulation to wait for the GDB debugger. This can be done via the command *-c system.cpu0.gdb_wait=1*. Alternatively, this property can be changed permanently in the `up.cfg` file. Now, the simulation, respectively the processor, waits for a remote connection with GDB. To set up the connection as easy as possible, the GDB debugger of the or1k-elf module should be used. After loading the module with *module load or1k-elf*, GDB can be started with the command *or1k-elf-gdb*. Additional useful features are the Text User Interface (TUI) and a gdbinit script `breakpoints.gdbinit`, in which breakpoints can be defined. These can be started by executing the command *or1k-elf-gdb -tui -x <file-path>/breakpoints.gdbinit*. Then the remote connection to the processor of the OR1kMVP can be initiated. Therefore, the command *target remote :55100* can be used. The port number 55100 is specified in the `up.cfg` file. Moreover, GDB should load the `.elf` file, so that it knows and can display e.g. the variable names. This has already been mentioned in section 2.6 on page 13 and can be done by using the command *symbol-file <file-path>/vmlinux-4.20.0.elf*. Additionally, *layout split* for the GDB TUI is recommended. Since the last three commands must be executed every time the simulation is started, it is a good idea to write them into the `breakpoints.gdbinit` file. Now, the remote connection between GDB and the processor of the OR1kMVP should be established and the simulation should continue automatically. In the following, breakpoints to functions of the Linux kernel can be set in GDB to pause the kernel and the simulation of the VP there and to investigate the kernel and the processor with GDB.

In addition to the kernel, the virtual hardware components should also be debugged in order to investigate their behavior. For this purpose, the debugger of the Eclipse IDE can be used. Eclipse is an open source programming tool for developing software in different programming languages. After the installation of Eclipse the code of the Virtual Components Modeling Library (VCML) and the OR1kMVP must be imported into the Eclipse development environment (IDE). Then, some environment variables have to be adjusted and a suitable compiler has to be selected. Finally, the code of the VP can be built, started and examined with the Eclipse debugger. The best way to do this is to create a new 'Launch Configuration' that defines `bin/or1kmvp` as the application and *-f config/up.cfg -c system.cpu0.gdb_wait=1* as the arguments. Now, the simulation can be started in Eclipse in 'Debug' mode and the GDB debugger can be started as described above. Then, breakpoints can be set in the code of the virtual hardware in Eclipse. If the simulation stops at a breakpoint in Eclipse, all variables and register contents can be displayed in the window 'Variables'. Even the program counter of the CPU can be read and in GDB a breakpoint in the kernel can be set there to see what the Kernel will do next.

3.3 Unit Testing and Google Test

A unit test (also called module test or component test) is used in software development to test individual parts of computer programs, i.e. to verify their correct functionality. Many software developers even go so far as to implement the unit tests before the actual code in order to define in advance what functionality the code should have. Then, they develop the code and optimize it until all unit tests run without errors. This is called test-driven software development. However, even if test-driven software development is not to be applied, it is advisable to write at least afterwards unit tests for a software project. This can especially help to test the basic functionality when modifying the code later, and to make sure that it was not corrupted by the modification of the code.

In the context of this work, the Google Test unit test framework is used [16]. Google Test basically is a unit test library for the programming language C++ . It offers a wide range of opportunities to expect values and function calls and to pre-define assertions. Additionally, Google Mock is used. This is an extension of the Google Test library which allows to define so-called C++ mock classes. A mock class is a class that 'pretends' to be another class. It is derived from the class it 'pretends' to be. The functions of the superclass can then be mocked. This means they are covered by mock functions. A mock function is actually not a real function, but only a dummy. The call of a mock function can be expected and if the call actually happens, it can execute predefined actions if required.
This way only the actual code can be tested and other classes that might be instantiated in this code are only simulated by mock classes.

4 The SDHCI Model

This chapter describes the SD Host Controller Interface (SDHCI) model and its implementation. Code sections and flowcharts are explained and many function calls are mentioned, especially in the initialization phase of the controller. The intention of this is that when the model is later modified by other people, suitable functions for setting breakpoints will directly stand out. Otherwise, the chapter will be structured chronologically according to the working steps in the development of the model. Therefore, it can also be used as a template for the development of other virtual hardware components.
All the code written in context of this Bachelor thesis and which is discussed in this chapter can be found on GitHub [4].

4.1 Device Tree Entry for the SDHCI Model

The first thing to do when implementing a new virtual component is to create a device tree entry (node) for it. As described in section 2.6 on page 13, the device tree is needed so that the Linux kernel knows all devices and can access them. Therefore, it is loaded and parsed early in the boot process.

```
sdhci1: mmc@99000000 {
    compatible = "fujitsu, mb86s70-sdhci-3.0";
    reg = <0x99000000 0x1000>;
    interrupts = <8>;
    bus-width = <1>;
    clocks = <&clock 2 2 0>, <&clock 2 3 0>;
    clock-names = "iface", "core";
};
```

Listing 4.1: Device tree entry for the SDHCI model

In Listing 4.1 the device tree entry for the SDHCI model is displayed. Before writing this entry, the documentation for SDHCI controllers by Fujitsu was studied. This can be found at `Documentation/devicetree/bindings/mmc/sdhci-fujitsu.txt` in the Linux source code [11].
In the header of the device tree node the name is specified first. Following, it is defined that it belongs to the family of MMC (Multi Media Card) devices and lastly, the bus address is specified, here 99000000. The most important thing about the device tree node is the compatible string in the first line. It tells the Linux kernel exactly which hardware device is present or to which hardware the current device is compatible. The Linux kernel uses this string to select the correct driver for the respective device. Which driver this is in the case of the SDHCI controller and how it is integrated into the Linux kernel is described in the next subsection 4.2. How the matching of the device and the right driver works in detail and how the compatible string is used in this process is described in subsection 4.5.2 on page 27.

The specifier `interrupts` defines the interrupt port number of the Central Processing Unit (CPU). It is absolutely necessary that this port number matches the port number in the processor model. More detailed information about the interrupt port is provided in section 4.3 on the next page. There, the new interrupt port of the CPU is defined.

In the last two lines of the device tree entry two clocks 'iface' and 'core' are specified, which must also be defined in the device tree. This is a requirement for all Fujitsu SDHCI controllers according to `Documentation/devicetree/bindings/mmc/sdhci-fujitsu.txt` in [11]. For this purpose, the following entry was added above the node for the SDHCI controller.

```
clock: oscillator{
    compatible = "fixed-clock";
   #clock-cells = <2>;
    clock-frequency = <50000000>;
    clock-output-names = "iface", "core";
};
```

Listing 4.2: Device tree entry for clocks needed for the SDHCI model

After changing the device tree source (`.dts` file) it has to be compiled again. Therefore, the device tree compiler (dtc) can be used, whose source code can be found in the Linux Git repository [11] at `scripts/dtc/`. Of course, it has to be build before using.
Then, the command *linux/BUILD/scripts/dtc/dtc -O dtb -o or1kmvp/sw/or1kmvp-up.dtb linux/arch/openrisc/boot/dts/or1kmvp-up.dts* was written into a 'prepare-release.sh' script. When the script is executed, the device tree compiler transforms the device tree source (`.dtsfile`) into a device tree blob (`.dtbfile`), which is readable for Linux. The device tree blob is stored in the software folder of the simulation, where the vmlinux-4.20.0, the vmlinux-4.20.0.elf, and the SD card image are also stored. For more information on the Linux Kernel and .elf files it should be referred to section 2.6 on page 13.

4.2 Driver for the SDHCI Model

As indicated in the previous section, the Linux kernel parses the device tree during the boot process to get to know all the devices of the system and to match the correct driver to each device. In order to be able to match the right driver, the driver has to be integrated to the Linux kernel first. This can be done in the kernel configuration. The default kernel configuration for OpenRISC processors can be found in [11] at `arch/openrisc/configs/or1kmvp_defconfig`. It was copied into the folder `linux/BUILD` as `.config`. Then, the configuration menu can be accessed by running the command *make -j4 ARCH=openrisc CROSS_COMPILE=or1k-elf- O=BUILD menuconfig* in a console from inside the folder `linux`.

In the configuration menu it should be navigated to the entries `Device Drivers`, then to `MMC/SD/SDIO card support` and finally to `Secure Digital Host Controller Interface support`. Now the key 'y' must be pressed to include this support into the kernel. After that, one can also add the `SDHCI support for Fujitsu Semiconductor F_SDH30`. An alternative way is to search `MMC_SDHCI_F_SDH30` in the search field of the configuration menu. After successful integration into the kernel, "[=y]" should appear behind this symbol.

After changing the kernel configuration, the Linux kernel has to be rebuild. In this case, it is called crosscompiling because the kernel is built on a powerful computer system for another Instruction Set Architecture (ISA). For this purpose, the command *make -j4 ARCH=openrisc CROSS_COMPILE=or1k-elf- O=BUILD all* can be used. The driver sdhci_f_sdh30 for the SDHCI controller is now finally built into the Linux kernel. If there is interest to have a look on the source code of the driver, in can be found in [11] in the directory `drivers/mmc/host`.

4.3 I/O Mapping and IRQ Mapping

After preparing the software (the Linux kernel), the implementation of the virtual hardware model and the integration of this into the Virtual Platform (VP) could start. How the result should look like was already shown in Figure 2.3 on page 12. First, the class `sdhci` was introduced. Afterwards, it was time to do the bindings in the code of the VP. This had to be done in the directories `or1kmvp/include/or1kmvp` and `or1kmvp/src/or1kmvp` in the files `system.h` and `system.cpp`.

```
 1 // additions to class system in system.h
 2
 3 public:
 4     vcml::property<vcml::range> sdhci;
 5
 6 private:
 7     vcml::generic::sdhci      m_sdhci;
 8     vcml::generic::sdcard     m_lasses_sdcard;
 9
10     sc_core::sc_signal<bool> m_irq_sdhci;
11
12
13 // additions in system.cpp
14
15 // to the initializer list of the constructor these lines are added
16     sdhci("sdhci", vcml::range(OR1KMVP_SDHCI_ADDR, OR1KMVP_SDHCI_END
           )),
17     m_sdhci("sdhci"),
18     m_lasses_sdcard("lasses_sdcard"),
19     m_irq_sdhci("irq_sdhci"),
20
21 // to the function body of the constructor these lines are added
22     m_sdhci.set_little_endian();
23
24     // I/O Mapping
25     m_bus.bind(m_sdhci.IN, sdhci);
26     m_sdhci.SD_OUT.bind(m_lasses_sdcard.SD_IN);
27
28     // IRQ Mapping
29     m_sdhci.IRQ.bind(m_irq_sdhci);
30     unsigned int irq_sdhci = cpu->irq_sdhci;      // in the for loop
31     cpu->IRQ[irq_sdhci].bind(m_irq_sdhci);        // in the for loop
```

Listing 4.3: Changes in `system.h` and `system.cpp` to include the SDHCI model to the VP

In lines 17 and 18, in the initializer list of the constructor, an SDHCI controller and an SD card are instantiated and initialized. These new virtual hardware components are then mapped in the body of the constructor. First, in line 25 the SDHCI device is bound to the bus, and one line below the SD card is bound to the SDHCI controller.

In line 29 the IRQ binding (interrupt signal lines) begins. The sc_signal m_irq_sdhci is connected on one side to the SDHCI controller and on the other side to the interrupt port of the processor(s). However, this interrupt port must first be created. Therefore, the interrupt port for the SDHCI controller was defined in `config.h` - in this case it is port number 8 (see Listing 4.4, line 8). In the description of the device tree in section 4.1 on page 21, it was already pointed out that the port number defined here has to match the port number in the device tree entry. Another important definition is the relative address (0x99000000) in line 3 that must also match the address in the device tree entry. Finally, in the initializer list of the OpenRISC processor the IRQ port for the SDHCI controller is instantiated (see line 18).

```
1  // additions in config.h
2
3  #define OR1KMVP_SDHCI_ADDR  (0x99000000)
4  #define OR1KMVP_SDHCI_SIZE  (OR1KISS_PAGE_SIZE)
5  #define OR1KMVP_SDHCI_END  (OR1KMVP_SDHCI_ADDR+OR1KMVP_SDHCI_SIZE-1)
6
7  // Interrupt map
8  #define OR1KMVP_IRQ_SDHCI   (8)
9
10 // additions in openrisc.h
11
12 public:
13     vcml::property<unsigned int> irq_sdhci;
14
15 // additions in openrisc.cpp
16
17 // to the initializer list of the constructor I added these line
18     irq_sdhci("irq_sdhci", OR1KMVP_IRQ_SDHCI),
```

Listing 4.4: Changes in `config.h`, `openrisc.h` and `openrisc.cpp`

Additionally, an SD card configuration in `or1kmvp/config/up.cfg` had to be added to tell the simulation which image to load for the SD card.

4.4 Simulated Hardware of the SD Host Controller Interface

In the beginning, the SDHCI model needed only three hardware interfaces to be connected to the VP. These are listed in Table 4.1. Since the three interfaces are all TLM sockets or SystemC out ports, in section 4.3 the `bind()` function could be used for mapping. The slave_socket IN, which is a TLM target socket, allows the CPU to access the SDHCI controller via the bus. "Behind" the hardware interface, there is the software interface. This is how the modeling of the individual registers is called. The processor can use the TLM socket and the offset addresses of the individual registers to write exactly where the modeled registers are located in the memory. On the following page, a list of all modeled registers of the SDHCI model with the corresponding offset addresses can be found. Note that adding Direct Memory Access (DMA) requires additional sockets and registers, which will be listed in section 4.6.

Port	**Type**	**Description**
IN	vcml::slave_socket *derived* from tlm_base_target_socket	Slave bus port
SD_OUT	vcml::sd_initiator_socket *derived* from tlm_base_initiator_socket	Master SD port
IRQ	vcml::out_port *derived* from sc_out<bool>	Interrupt port

Table 4.1: Hardware interfaces

Name	**Offset**	**Access**	**Width**	**Description**
BLOCK_SIZE	+0x004	RW	16 bit	Block Size Register
BLOCK_COUNT_16BIT	+0x006	RW	16 bit	16 bit Block Count Register
ARG	+0x008	RW	32 bit	Argument Register
TRANSFER_MODE	+0x00C	RW	16 bit	Transfer Mode Register
CMD	+0x00E	RW	16 bit	Command Register
RESPONSE	+0x010	ROC	4*32 bit	Response Register
BUFFER_DATA_PORT	+0x020	RW	32 bit	Buffer Data Port Register
PRESENT_STATE	+0x024	RO	32 bit	Present State Register
HOST_CONTROL_1	+0x028	RW	8 bit	Host Control 1 Register
POWER_CTRL	+0x029	RW	8 bit	Power Control Register
CLOCK_CTRL	+0x02C	RW	16 bit	Clock Control Register
TIMEOUT_CTRL	+0x02E	RW	8 bit	Timeout Control Register
SOFTWARE_RESET	+0x02F	RWAC	8 bit	Software Reset Register
NORMAL_INT_STAT	+0x030	RW1C	16 bit	Normal Interrupt Status Register
ERROR_INT_STAT	+0x032	RW1C	16 bit	Error Interrupt Status Register
NORMAL_INT_STAT_ENABLE	+0x034	RW	16 bit	Normal Interrupt Status Enable Register
ERROR_INT_STAT_ENABLE	+0x036	RW	16 bit	Error Interrupt Status Enable Register
NORMAL_INT_SIG_ENABLE	+0x038	RW	16 bit	Normal Interrupt Signal Enable Register
ERROR_INT_SIG_ENABLE	+0x03A	RW	16 bit	Error Interrupt Signal Enable Register
CAPABILITIES	+0x040	HWInit	2*32 bit	Capabilities Register
MAX_CURR_CAP	+0x048	HWInit	32 bit	Maximum Current Capabilities Register
HOST_CONTROLLER_VERSION	+0x0FE	HWInit	16 bit	Host Controller Version Register
F_SDH30_AHB_CONFIG	+0x100	RW	16 bit	Controller Specific Register
F_SDH30_ESD_CONTROL	+0x124	RW	32 bit	Controller Specific Register

Table 4.2: Software interfaces

All registers except the last two are described in the *SD Host Controller Simplified Specification* (refer to [13]). The last two are controller specific registers. The *F_SDH30_AHB_CONFIG* register provides the configuration for the communication via the Advanced High-performance Bus (AHB) and the *F_SDH30_ESD_CONTROL* register indicates that there is no embedded SD card (eSD). These registers are just read and written during the initialization. They are not necessary for the further functionality of the SDHCI model.

4.5 Communication of the Host and the SD-Card via SDHCI

This subchapter deals with the functionality of the SDHCI. It explains how the operating system (OS) sends commands to the SDHCI model. Additionally, it describes the initialization process of the controller and the read and write process on the SD card via the SDHCI.

4.5.1 Sending Commands to the SD Card

Sending a command to the SD card via the SDHCI model is always the same procedure regardless of the command. The SDHCI driver can only communicate with the SDHCI controller via the registers of the software interface, which are listed in Table 4.2 on the previous page. When the host wants to send a command to the SD card, it always writes the argument into the 32 bit *Argument* register of the SDHCI controller first. Then, it writes the command into the 16 bit *Command* register. In the *Command* register the bits D13-D08 are provided for the command index. Bits D07-D00 are intended for flags of the command, e.g. bit D04 indicates whether the controller should check the command index in the response of the SD card or not. The other bits are reserved and must not be written. If, for example, CMD18 is to be sent and the index is to be checked, the value 0x1210h (0x12h = 18) would be written into the *Command* register. The *Command* register is always the last register the host should write, because the controller starts processing the command immediately after writing this register. This is achieved via the write function of the register. For this purpose, in the constructor of the SDHCI model it is specified that the *Command* register can be read and written (see Listing 4.5, line 1). Here is also defined that the function `sdhci::write_CMD()` is called when writing to this register.

```
1 CMD.allow_read_write();
2 CMD.read = &sdhci::read_CMD;
3 CMD.write = &sdhci::write_CMD;
```

Listing 4.5: Parts of the contructor `sdhci()`

Obviously, the function `sdhci::write_CMD()` is central for the processing of commands, thus it is helpful to have a closer look at it (Listing 4.6 on the next page).

In line 2 of the code a function is called which sets the **Command Inhibit (CMD)** bit in the *Present State* register. This indicates that there is currently a communication on the "Command Line" between the SDHCI model and the SD card model. This "Command Line" is not modeled as an sc_signal, but it makes sense to display it in the *Present State* register. Thus, it can be always seen from the outside (through the *Present State* register) in which phase the controller is at the moment. Additionally, when the **Command Inhibit (CMD)** bit in line 16 is cleared, the **Command Complete** Interrupt is automatically written to the *Interrupt* register. But, note that the hardware interrupt is only triggered in line 35.

In lines 4 to 8 the values of the *Command* and the *Argument* register are read and put into the sd_command m_cmd. This is then transported to the SD card in line 12. The SD card processes the command and returns an sd_status, e.g. SD_OK. The m_cmd is passed call-by-reference, so the SD card can write the response directly into the data member response, which is a 17 byte array. The SDHCI controller writes the response into the *Response* registers in line 14. The position within the *Response* registers depends on the type of response. See the implementation of the function `write_sd_resp_to_RESPONSE()` or [13, p. 48].

Starting in line 18, the returned status of the SD card is evaluated. SD_OK_TX_RDY means that the SD card is ready for reading and that it has already loaded the requested data into its internal buffer. SD_OK_RX_RDY means that the write operation can begin.

```
u16 sdhci::write_CMD(u16 val) {
    set_PRESENT_STATE(COMMAND_INHIBIT_CMD);

    m_cmd.spi = false;
    m_cmd.opcode = (val & 0x3F00) >> 8;
    m_cmd.argument = ARG;
    m_cmd.crc = calc_crc7();
    m_cmd.resp_len = 0;

    [...]

    m_status = SD_OUT->sd_transport(m_cmd);

    write_sd_resp_to_RESPONSE();

    set_PRESENT_STATE(~COMMAND_INHIBIT_CMD);

    switch (m_status) {
    case SD_OK:
        break;
    case SD_OK_TX_RDY:
        transfer_data_from_sd_buffer_to_sdhci_buffer();
        set_PRESENT_STATE(BUFFER_READ_ENABLE);
        break;
    case SD_OK_RX_RDY:
        set_PRESENT_STATE(BUFFER_WRITE_ENABLE);
        break;

    [...]

    }

    [...]

    IRQ.write(true);
    return val;
}
```

Listing 4.6: Parts of the code of the function `sdhci::write_CMD()`

4.5.2 Initialization Process

The initialization of the SDHCI controller and the SD card are two separate phases, which should be discussed more in detail. As described in sections 4.1 to 4.3, the Linux OS now knows the hardware component SDHCI and the driver sdhci_f_sdh30. After the CPU is initialized and running, it starts to initialize the peripheral devices. This is done in the function `do_basic_setup()` in `init/main.c` [11]. From there, initcall functions are called for all loaded driver modules. So, the function `sdhci_f_sdh30_driver_init()` is called, which is located in the driver `drivers/mmc/host/sdhci_f_sdh30.c` [11]. The host then iterates over all peripherals attached to the bus and tries to match the device with the current driver sdhci_f_sdh30.

This iteration is done in `bus_for_each_dev()` in `drivers/base/bus.c` [11]. Therein, for each device the function `platform_match(drv, dev)` in `drivers/base/platform.c` [11] is called. This function again calls `of_driver_match_device()` and here finally a string comparison is used to find out whether the current device (the compatible string) and the driver (the f_sdh30_dt_ids.compatible) match. If so, the probe function of the driver is called. For the driver sdhci_f_sdh30 the probe function is `sdhci_f_sdh30_probe()`.
In this probe function registers of the SDHCI controller are read and written for the first time, namely the controller specific registers *F_SDH30_AHB_CONFIG* and *F_SDH30_ESD_CONTROL*. Finally, the function `sdhci_setup_host()` is called, that again calls `__sdhci_read_caps()`. This resets the SDHCI controller by writing in the *Software Reset* register and finally, the *Host Controller Version* and *Capabilities* register are read.

All these steps are also shown in the following Flowchart 4.1. They complete the initialization of the SDHCI controller because the host system now knows everything about it.

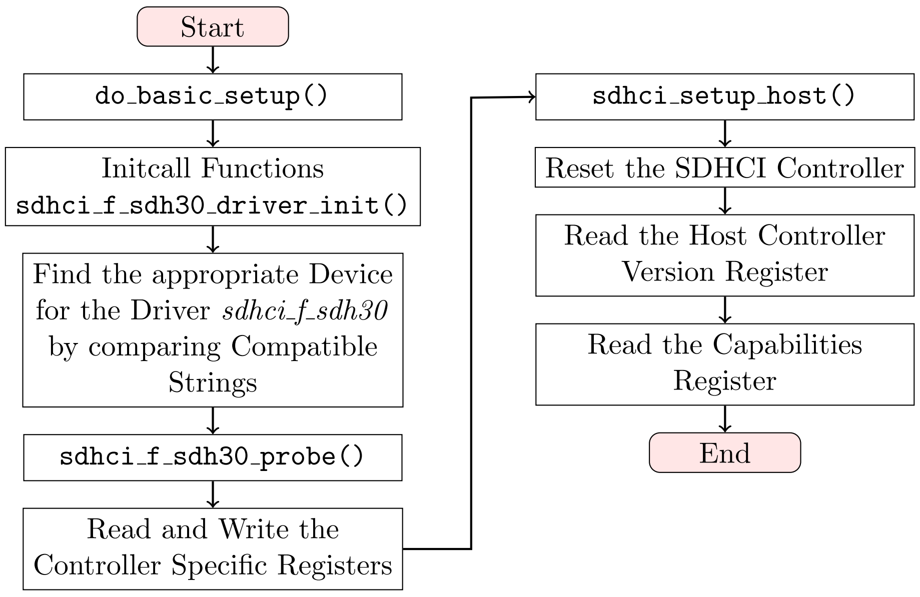
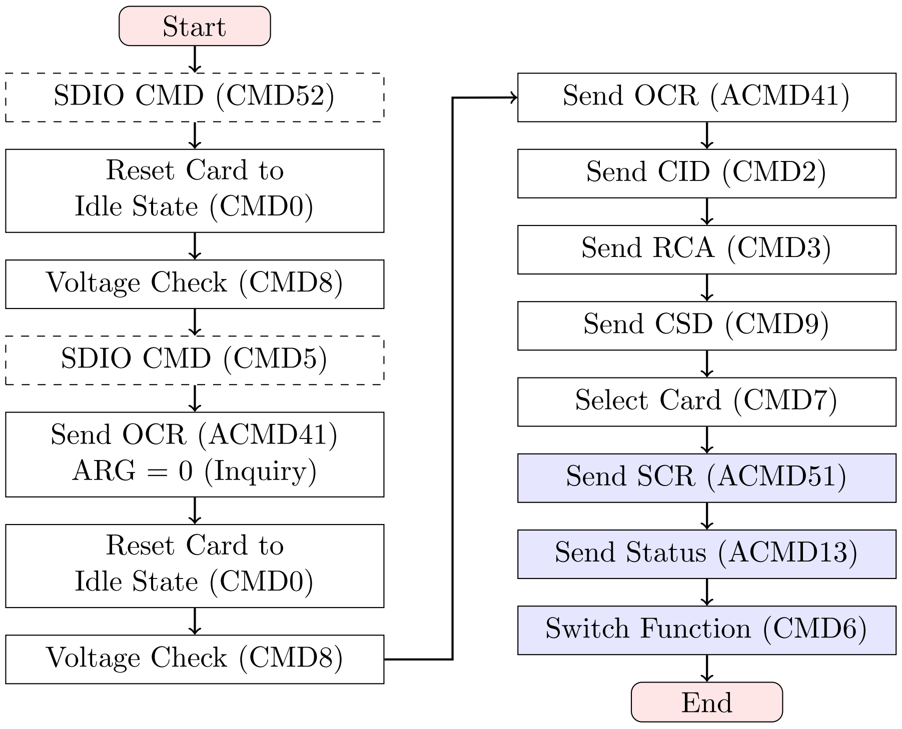

Start
do_basic_setup()
Initcall Functions sdhci_f_sdh30_driver_init()
Find the appropriate Device for the Driver *sdhci_f_sdh30* by comparing Compatible Strings
sdhci_f_sdh30_probe()
Read and Write the Controller Specific Registers
sdhci_setup_host()
Reset the SDHCI Controller
Read the Host Controller Version Register
Read the Capabilities Register
End

Figure 4.1: Initialization process of the SDHCI controller

The second phase of the initialization is the initialization of the SD card. Therefore, a sequence of commands is sent to the SD card. This sequence is shown in Flowchart 4.2.

For more information about each command, please see [17, p. 86 f.]. The sequence of the commands is defined in [13, p. 160 f.] and is executed in `mmc_rescan_try_freq()`.

The last three command blocks in Figure 4.2 are colored blue because they already use the read and write functions, which are introduced principal in the next subsection 4.5.3. The dashed command blocks are sent, but the SD card cannot execute them, because they are SDIO commands that it does not support. An introduction to SDIO and SDIO commands is provided in subsection 2.7.



Legend

OCR	-	Operation Condition Register	CID	-	Card ID
RCA	-	Relative Card Address	CSD	-	Card Specific Data
SCR	-	SD Configurations Register			

Figure 4.2: Initialization process of the SD card

After sending all these commands, the initialization of the SDHCI device and the SD card is completed and it is possible to start reading and writing data blocks from or to the SD card.

4.5.3 Reading Data from the SD Card and Writing Data to the SD Card

Reading and writing on the SD card is command based. There are two commands that initiate reading from memory, command 17 (CMD17 - read single block) and command 18 (CMD18 - read multiple block). On the other hand, there are command 24 (CMD24 - write single block) and command 25 (CMD25 - write multiple block) which tell the SD card that something should be written to its memory. It can be seen that the memory of the SD card is divided into blocks of the same size, in the case of the SD card in the VCML library it is a block size of 512 bytes. Before one of the described four commands can be sent to the SD card, the SDHCI controller is told how big the data blocks are and how many blocks should be transferred. Therefore, the *Block Size* register, the *Block Count* register, and the *Transfer Mode* register are written first. Then, the *Argument* and the *Command* register can be written and the controller performs all its tasks as described in subsection 4.5.1 on page 26. In the *Argument* register, the offset address is specified at which the data is to be read or written on the SD card.
At this point, it is referred to the upper part of Figure 4.3 on the next page, where the steps described here are shown again. The green boxes represent the actions that are executed for every command and described in subsection 4.5.1. Also note that this flowchart shows the actions of the host driver and not those of the SDHCI controller. The flowchart was created by using Figure 3-13 from [13], the driver's code, and debugging experience.

When the SD card has processed the command and is ready for read or write access, it returns

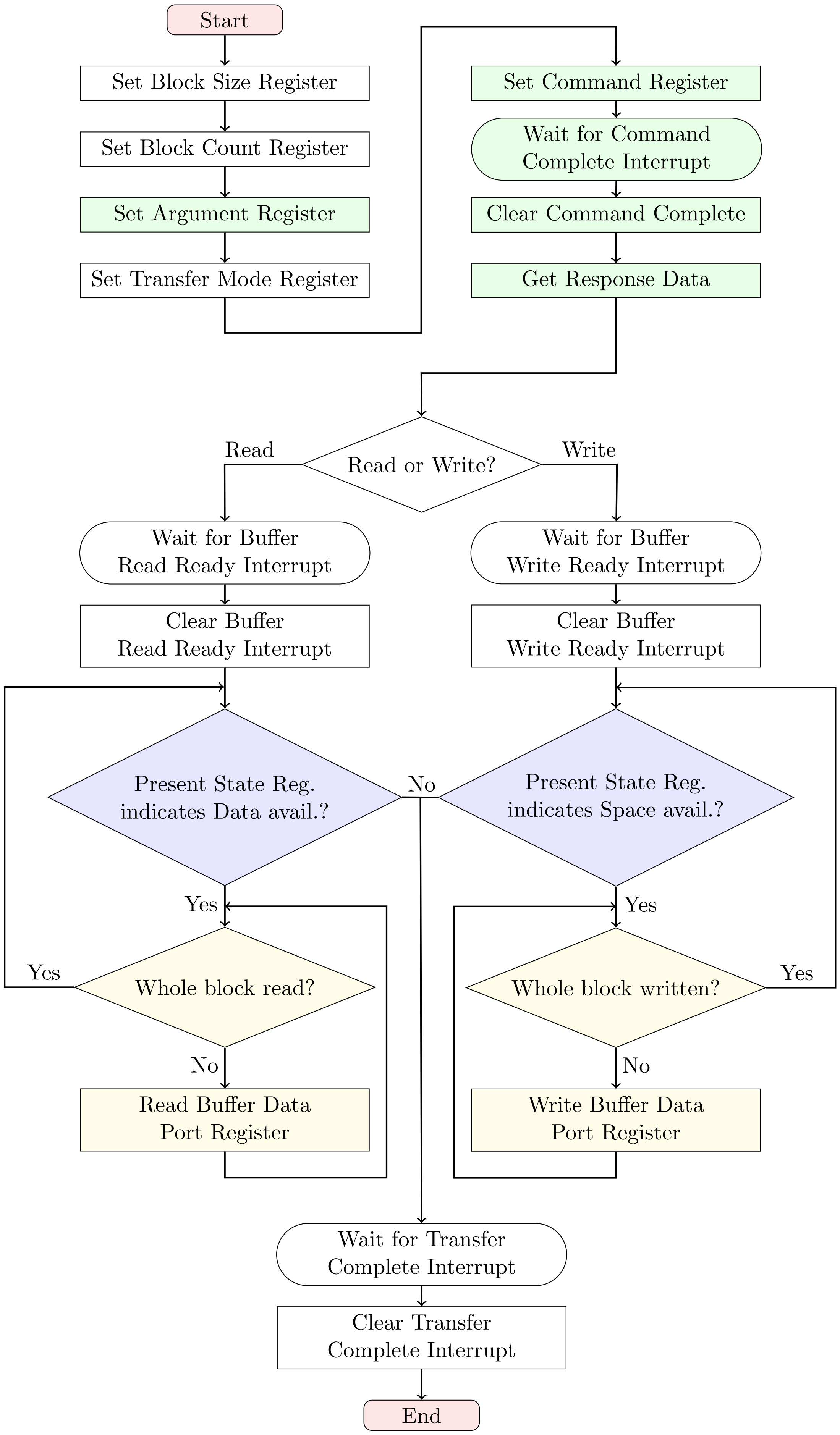

Start
Set Block Size Register
Set Block Count Register
Set Argument Register
Set Transfer Mode Register
Set Command Register
Wait for Command Complete Interrupt
Clear Command Complete
Get Response Data
Read or Write?
Read
Write
Wait for Buffer Read Ready Interrupt
Clear Buffer Read Ready Interrupt
Wait for Buffer Write Ready Interrupt
Clear Buffer Write Ready Interrupt
Present State Reg. indicates Data avail.?
No
Present State Reg. indicates Space avail.?
Yes
Whole block read?
Yes
No
Read Buffer Data Port Register
Yes
Whole block written?
Yes
No
Write Buffer Data Port Register
Wait for Transfer Complete Interrupt
Clear Transfer Complete Interrupt
End

Figure 4.3: Reading and writing from or to the SD card via SDHCI not using DMA

the status SD_OK_TX_RDY or SD_OK_RX_RDY. The former means that the read operation can begin and that the SD card has already loaded the requested block into its internal buffer, which has exactly the size of a block plus two CRC bytes. As shown in Listing 4.6 on page 27 in line 22, the SDHCI device then receives this buffer byte by byte and stores it to its own 514 byte internal buffer. In addition, the **Buffer Read Enable** bit is set in the *Present State* register. When writing this bit, the **Buffer Read Ready** Interrupt is automatically written into the *Interrupt* register. If the SD card returns the status SD_OK_RX_RDY, the **Buffer Write Enable** bit is set in the *Present State* register and the **Buffer Write Ready** Interrupt is written to the *Interrupt* register.

So, it is comprehensible that in line 35 of Listing 4.6 not only the **Command Complete** Interrupt but also the **Buffer Read/Write Ready** Interrupt is triggered. As shown in the flowchart, the driver first processes the **Command Complete** Interrupt and then the **Buffer Read/Write Ready** Interrupt. When processing the latter, the function `sdhci_transfer_pio()` in `drivers/mmc/host/sdhci.c` [11] is called, regardless of whether it is a read or write transaction. It is very worthwhile to have another look at this function.

The function `sdhci_transfer_pio()` has a loop that reads (`sdhci_read_block_pio()`) or writes (`sdhci_write_block_pio()`) individual blocks as long as the *Present State* register indicates that data (**Buffer Read Enable** bit) or space (**Buffer Write Enable** bit) is available. Note that contrary to Figure 3-13 in [13], there is no need to wait for an interrupt after each block. Instead, the read and write operation is performed just based on the information in the *Present State* register. The head of the loop in the function `sdhci_transfer_pio()` is shown in blue in the Figure 4.3 on the previous page. The functions `sdhci_read_block_pio()` and `sdhci_write_block_pio()` called by it are shown in yellow.

In the following, the reading process will be discussed first. The data transfer from the SDHCI controller to the host system is done via the *Buffer Data Port* register, which is 4 bytes in size. The function `sdhci_read_block_pio()` first initializes its own count variable for the bytes in order to be able to recognize when a block was read completely. Then it starts reading the *Buffer Data Port* register. Thereby, in the SDHCI model the function `read_BUFFER_DATA_PORT()` is called, which is shown in Listing 4.7. This is done similar to the *Command* register in section 4.5.1. In this function, the first (or the next) four bytes are copied from the SDHCI buffer to the *Buffer Data Port* register (line 3). This is done using a buffer pointer that always points to the next byte to be read. In line 5 it is checked whether the block has been read completely. If so, the value of the *Block Count* register is decremented and the buffer pointer is set back to the beginning of the buffer. Then, it is checked whether all data blocks have already been read (line 10). If this also applies, the *Present State* register must indicate that the read operation has been completed. For this purpose, the **Data Line Active** and **Read Transfer Active** bits are cleared. This also triggers the **Transfer Complete** Interrupt. If not all blocks have been read, the next block is transferred from the SD card's buffer to the SDHCI buffer. Afterwards, it is indicated in the *Present State* register that the read operation can continue (**Buffer Read Enable** bit).

Writing to the SD card also works via the *Buffer Data Port* register. Therefore, the next four bytes are written into this register by the host driver. This is done in the function `sdhci_write_block_pio()`. During this write operation the function `write_BUFFER_DATA_PORT(u32 val)` in the SDHCI model is called, which is shown in Listing 4.8. This function copies the four bytes from the *Buffer Data Port* register into the SDHCI buffer. If the buffer is full, i.e. a complete block has been sent to the SDHCI controller, the CRC code for the block is calculated and the block is sent byte by byte to the SD buffer. If all blocks are written (line 14), this is displayed in the *Present State* register and the **Transfer Complete** Interrupt is triggered.

```
u32 sdhci::read_BUFFER_DATA_PORT() {
    if (PRESENT_STATE && BUFFER_READ_ENABLE) {
        transfer_data_from_sdhci_buffer_to_BUFFER_DATA_PORT();

        if (m_bufptr >= (BLOCK_SIZE & 0x0FFF)) {     // one block read
            set_PRESENT_STATE(~BUFFER_READ_ENABLE);
            BLOCK_COUNT_16BIT -= 1;
            m_bufptr = 0;

            if (BLOCK_COUNT_16BIT == 0) {           // all blocks read
                set_PRESENT_STATE(~READ_TRANSFER_ACTIVE);
                IRQ.write(true);
                return BUFFER_DATA_PORT;
            }
            transfer_data_from_sd_buffer_to_sdhci_buffer();
            set_PRESENT_STATE(BUFFER_READ_ENABLE);
        }
        return BUFFER_DATA_PORT;
    }
    [...]
}
```

Listing 4.7: Parts of the code of the function `sdhci::read_BUFFER_DATA_PORT()`

```
u32 sdhci::write_BUFFER_DATA_PORT(u32 val) {
    if (PRESENT_STATE && BUFFER_WRITE_ENABLE) {
        BUFFER_DATA_PORT = val;
        transfer_data_from_BUFFER_DATA_PORT_to_sdhci_buffer();

        if (m_bufptr >= (BLOCK_SIZE & 0x0FFF)) { // one block written
            set_PRESENT_STATE(~BUFFER_WRITE_ENABLE);
            BLOCK_COUNT_16BIT -= 1;
            m_bufptr = 0;
            [...] // generate two bytes CRC code

            transfer_data_from_sdhci_buffer_to_sd_buffer();

            if (BLOCK_COUNT_16BIT == 0) {        // all blocks written
                set_PRESENT_STATE(~WRITE_TRANSFER_ACTIVE);
                IRQ.write(true);
                return val;
            }
            set_PRESENT_STATE(BUFFER_WRITE_ENABLE);
        }
        return val;
    }
    [...]
}
```

Listing 4.8: Parts of the code of the function `sdhci::write_BUFFER_DATA_PORT()`

4.6 DMA Functionality for the SDHCI Model

The next step in the development of the SDHCI model is the introduction of DMA - more precisely Single Operation DMA (SDMA). As introduced in section 2.8 on page 15, DMA means that a peripheral device - in this case the SDHCI device - can directly access the main memory. This access can be read or write access. As explained in section 2.8 the CPU passes an address of the main memory to the SDHCI. At this address the SDHCI controller can read or write data autonomously. Thus, the CPU does not have to transmit all data to the SDHCI via the *Buffer Data Port* register anymore, but the SDHCI can read or write it block by block from or to the main memory while the CPU accomplishes other tasks. The resulting speedup of the communication and the whole VP is documented in chapter 5.

To enable the CPU to pass the DMA address to the SDHCI controller, a new register must be added to the SDHCI. This is the *SDMA System Address* register which is located at offset address 0 of the SDHCI device and is 32 bit wide. Additionally, the SDHCI model must get a new vcml::master_socket at the hardware interface level, which was explained in section 2.4. This is necessary for the SDHCI to be able to write onto the bus and thus into the memory attached to it. These are the only two changes to the hardware and software interfaces that become necessary with the introduction of DMA for the SDHCI model.

The next step in the introduction of DMA is to examine what is changing on the driver's side and what other actions it is taking compared to the normal Peripheral Input/Output (PIO) communication shown in Figure 4.3. This can be seen in Figure 4.4 on the next page. It was created using Figure 3-14 [13]. There, it is recognizable that at the beginning of the command initiation is not a lot changing. Only, the first operation block has been added, which shows that the driver, respectively the CPU, has to write the *SDMA System Address* register first. The other commands are identical to the communication without DMA until the last green block (see Figure 4.3). Remember that the green blocks are identical for all commands - even those without read and write transfer.

In addition, a vcml::property DMA_enabled is added to the SDHCI model to easily enable and disable the DMA functionality of the model. This property is initialized by default with true. Moreover, when reading the *Capabilities* register the appropriate register value is returned depending on this property. This way, the Linux driver knows if DMA should be used or not. The second important point where this property is used is in the evaluation of the status the SD card returns. In Listing 4.6 this happens in lines 18 to 27. If the SD card is ready for a read (SD_OK_TX_RDY) or a write (SD_OK_RX_RDY) operation, then usually the appropriate SDHCI status and thus the appropriate interrupt is generated to initiate the read or write operation. With the introduction of DMA, the SDHCI controller is to take over this completely. Therefore, a check is introduced in the described two cases whether DMA should be used or not. This is also done via the vcml::property DMA_enabled. If DMA is to be used, the **Data Line Active** bit in the *Present State* register is set and afterwards a newly introduced event is notified. This event is called m_start_DMA_event and indicates that a DMA transaction should start. An sc_thread is waiting for this event. This sc_thread is named `exec_DMA_transfer()` and shown in Listing 4.9 on page 35. It runs since the beginning of the SystemC simulation. If more information about the SystemC simulation or sc_threads is needed, it is recommended to have a look at section 2.2 on page 6.

In Listing 4.9 it can be seen that in line 6 of the thread `exec_DMA_transfer()` it is waited for the event m_start_DMA_event described above. As soon as this event is notified, a DMA transaction is executed. First of all, the upper bits of the *Block Size* register are used to calculate a DMA boundary. This is explained in section 2.8 on page 15. However, the SDHCI cannot exceed this boundary, since with 512 kB it is very large and the Linux driver ensures that it never requests such large amounts of data. Nevertheless, a mechanism has been im-

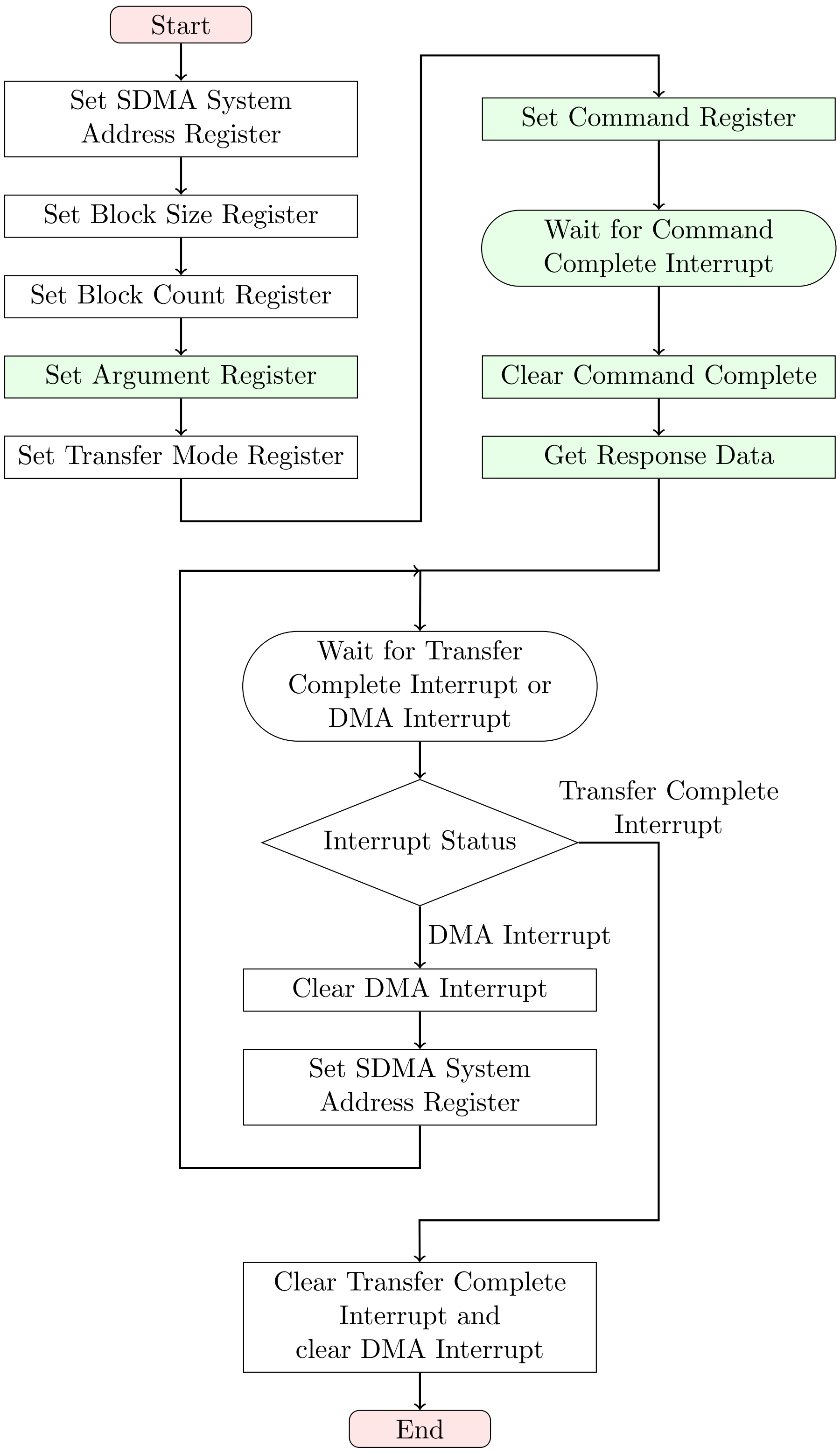

Start
Set SDMA System Address Register
Set Block Size Register
Set Block Count Register
Set Argument Register
Set Transfer Mode Register
Set Command Register
Wait for Command Complete Interrupt
Clear Command Complete
Get Response Data
Wait for Transfer Complete Interrupt or DMA Interrupt
Interrupt Status
Transfer Complete Interrupt
DMA Interrupt
Clear DMA Interrupt
Set SDMA System Address Register
Clear Transfer Complete Interrupt and clear DMA Interrupt
End

Figure 4.4: Reading and writing from or to the SD card via SDHCI using DMA

plemented to detect when the boundary is exceeded. Then, an error message is thrown, so that the behavior of the SDHCI in this case can be implemented later.

```
 1 void sdhci::exec_DMA_transfer() {
 2     u32 boundary = 0;
 3     tlm_response_status rs;
 4 
 5     while (true) {
 6         wait(m_start_DMA_event);
 7 
 8         boundary = ((4096 << ((BLOCK_SIZE & 0x7000) >> 12)) - 12);
 9 
10         if (m_status == SD_OK_TX_RDY) {
11             rs = DMA_read_from_sd(boundary);
12         } else if(m_status == SD_OK_RX_RDY) {
13             rs = DMA_write_to_sd(boundary);
14         } else {
15             VCML_ERROR("DMA for this command not permitted!");
16         }
17 
18         if (failed(rs))
19         VCML_ERROR("DMA by the SDHCI controller failed");
20 
21         set_PRESENT_STATE(~DAT_LINE_ACTIVE);
22         NORMAL_INT_STAT |= INT_TRANSFER_COMPLETE;
23         IRQ.set();
24     }
25 }
```

Listing 4.9: Thread `exec_DMA_transfer()` that executes the DMA transfer

After the boundary has been calculated, it is checked whether the DMA transfer is a read or write operation (lines 10 and 12). If it is a read operation, the function `DMA_read_from_sd()` is called (line 11). If in contrast something is to be written to the SD card, the function `DMA_write_to_sd()` is called (line 13). The calculated boundary is passed as parameter to both functions. The functions `DMA_read_from_sd()` and `DMA_write_to_sd()` execute the DMA operations by accessing the vcml::memory via the newly added vcml::master_socket.
When reading from the SD card the function `OUT.write(SDMA_SYSTEM_ADDRESS, m_buffer, (BLOCK_SIZE & 0x0FFF))` is used. It writes the whole contents of the SDHCI buffer to the address specified in the *SDMA_System_Address* register. Then, the value of this register is incremented by the block size and a new block is loaded from the SD card into the SDHCI buffer. This is repeated until all blocks are read from the SD card and loaded into the main memory using DMA.
The write process works very similar. In the function `DMA_write_to_sd()` a whole block (512 bytes) is read with the function `OUT.read(SDMA_SYSTEM_ADDRESS, m_buffer, (BLOCK_SIZE & 0x0FFF))` and stored in the SDHCI buffer. The contents of the SDHCI buffer is then transferred to the SD card. This is repeated until all data has been written to the SD card.

When the DMA data transfer is completed in one of the two explained functions, in line 18 of Listing 4.9 it is checked whether the transaction was successful. Subsequently, the simulated ”Data Line” is released and the **Transfer Complete** Interrupt is triggered. In Figure 4.4 on page 34 it can be seen that the SDHCI driver is waiting for this interrupt. It completes the DMA transfer.

4.7 Google Test for the SDHCI Model

As described in section 3.3, it is necessary for the maintainability of the SDHCI model to implement unit tests that test the functionality of the code. Therefore, a Google Test unit test for the SDHCI model was implemented which can be found on GitHub in `test_sdhci.cpp` [4]. In the Google Test there are seven test scenarios, three testing the SDHCI without DMA and four testing it with DMA functionality. In the following the main skeleton of the SDHCI test and all the test cases will be explained.

At the beginning of the Google Test `test_sdhci.cpp`, two classes are defined which are needed to build the test bench. The class initiator simulates the entire VP, i.e. the CPU, the memory, and the system bus. Therefore, it has a vcml::master_socket to which the SDHCI controller is connected, as usually to the bus. Moreover, it contains a vcml::generic::memory to which the vcml::master_socket of the SDHCI controller is connected. This becomes important only when testing the DMA functionality in the last four test cases.
The second class implemented in the Google Test for the SDHCI model is the class mock_sdcard. It mocks the SD card connected to the vcml::sd_initiator_socket of the SDHCI. Mocking is explained in section 3.3 on page 19. Mocking the SD card is necessary to avoid instantiating a complete SD card and to avoid loading the large Linux image every time. To simulate the behavior of the SD card three functions of the SD card, that the SDHCI calls, are mocked. The three mocked functions can be seen in Listing 4.10.

```
MOCK_METHOD1 (sd_transport, vcml::sd_status(vcml::sd_command&));
MOCK_METHOD1 (sd_data_read, vcml::sd_tx_status(vcml::u8&));
MOCK_METHOD1 (sd_data_write, vcml::sd_rx_status(vcml::u8));
```

Listing 4.10: Mocked functions of the SD card

The first thing done inside the `Test(sdhci, sdhci)` function is the instantiation of the models and the binding. Then, the individual test cases are performed. The vcml::property sdhci.DMA_enabled is always used to determine whether the test case must be performed with or without DMA functionality. In addition, the SDHCI controller is reset by writing to the *Reset* register before each new test case. The reason why not every test case gets its own test function is that at the end of a test the SystemC simulation would be stopped and thus the sc_thread of the SDHCI model would be killed. Afterwards, the SystemC simulation and the thread cannot be started manually anymore, so the way of sequential implementation of the different cases in a single test is the best solution.

In the first test case, a command 0 (CMD0) is sent to reset the SD card to idle state. Then, the test expects a call of the function `sd_transport()`, via which the SDHCI model forwards the command to the SD card. This is one of the mocked functions, thus the return statement of the SD card is pre-defined. Moreover, it is checked whether the SDHCI correctly writes the response to the *Response* register and whether the correct interrupt is triggered.

The second test case initiates a multiple block read transaction. Two blocks of size 8 bytes are to be read, which means that the *Buffer Data Port* register must be read four times. In this test case the mocked function `sd_data_read()` plays an important role because it is called 16 times by the SDHCI controller. In the further course of the test case the values of the *Buffer Data Port* register and all interrupts are checked.

In the third test case the multiple block write functionality is tested, i.e. two blocks of size 8 bytes are written to the mocked SD card. For this the mocked function `sd_data_write()` is called 16 times. The verification of the written data and the interrupts is similar to that in test case two.

The last four of the seven test cases in the Google Test `test_sdhci.cpp` test the DMA functionality of the SDHCI model. Test cases number four and five test the reading and writing of multiple blocks via DMA. These tests work similar to the reading and writing tests without DMA. A transaction is initiated in which the SDHCI either receives data from the SD card to write to the test memory or reads data from the test memory to forward it to the SD card.
Test cases number six and seven are negative tests. This means that the SDHCI controller is deliberately brought into an error state and it is checked whether it issues an error message and cancels the transaction. The two most important error states are checked. The first test triggers an exceeding of the SDMA boundary and checks whether an error message - a VCML_ERROR - is thrown. This should never happen in the real application, because the Linux driver makes sure that it only initiates the transaction of small amounts of data. However, testing the error message makes sense simply because no mechanisms have been implemented that could handle a boundary overflow. The second negative test tries to write the *Buffer Data Port* register while the SDHCI controller is in DMA mode. This is not allowed because the data transaction can only be done either via DMA access or via writing the *Buffer Data Port* register. Both may never happen at the same time.

5 Experimental Evaluation - Results

After the SD Host Controller Interface (SDHCI) has been implemented and extended with Direct Memory Access (DMA) functionality, now of course it should be tested what advantage this has brought for the Virtual Platform (VP). So, it shall be examined whether there is a performance increase by the SDHCI model and how big it is. This naturally raises the question of what is a good benchmark for the performance of the OpenRISC 1000 Multicore Virtual Platform (OR1kMVP) and how to measure it. Here, two approaches shall be proposed and applied.

5.1 Speedup in the Read and Write Performance

The first approach tests how fast the communication between the memory of the VP and the SD card works. The idea is that the CPU should load a large amount of data from the SD card into the memory or request loading it by the SDHCI controller. The duration of this loading process is then to be measured. For the measurement of the read and write performance, it does not matter that only a read operation from the SD card is examined here, since the same actions are performed in reverse order for reading and writing. Therefore, read and write operations are equally fast.

To do the measurement, it is necessary to first mount the image of the SD card in order to change it. Mounting means to convert the SD card image back into a file system on a host. Then, it can be changed and afterwards it can be converted into an image file again. So, the SD card image was mounted and a large amount of data was stored on it. In this case, a file named `random_file` with random data of the size 10 MB was created. Now, the correct place and the correct function had to be found, which causes the loading of this large amount of data into the main memory. The function `md5sum` appeared to be the best solution, which calculates the MD5 checksum of a file. The optimal point to calculate this checksum automatically is after the Linux boot and just before opening the user console. Here, the init file, which can be found on the SD card in `/etc/init.d/rcS`, is a good choice. The code in this file is executed once immediately before opening the console. Here, the command `measure md5sum random_file` was added. The command `measure`, whose code is located in the `bin` folder on the SD card, measures the execution time of the command `md5sum` by simply subtracting the runtime values before and after the execution. The command `md5sum random_file` calculates the MD5 checksum of the 10 MB random data. Afterwards, the output of the `measure` function is printed in the console. This time value contains the time for loading the SD card into the main memory and the time for the mathematical calculation of the checksum. The latter makes up a much smaller part and is therefore negligible. After the calculation of the `md5sum` the simulation is automatically terminated.
With this first method of performance measurement the speedup in the communication between the main memory and the SD card can be determined. The following Figure 5.1 depicts the measured times for calculating the `md5sum`. It can be seen that the communication with the SD card via the SDHCI controller is significantly faster than via SPI, as Jan Henrik Wein-

stock had implemented it. Additionally, the implementation of DMA brought a performance gain. Table 5.1 shows the average times for the calculation of the `md5sum`. From this, it can be calculated that PIO transfer is $\frac{190.8553sec}{58.4422sec} = 3.27$ times as fast as the communication via SPI. Remember that the PIO communication is the communication via the SDHCI without DMA. Moreover, the DMA transfer is $\frac{58.4422sec}{42.5469sec} = 1.37$ times as fast as the communication via PIO. In total, the communication via DMA is 4.5 times as fast as the original communication with the SD card via SPI.

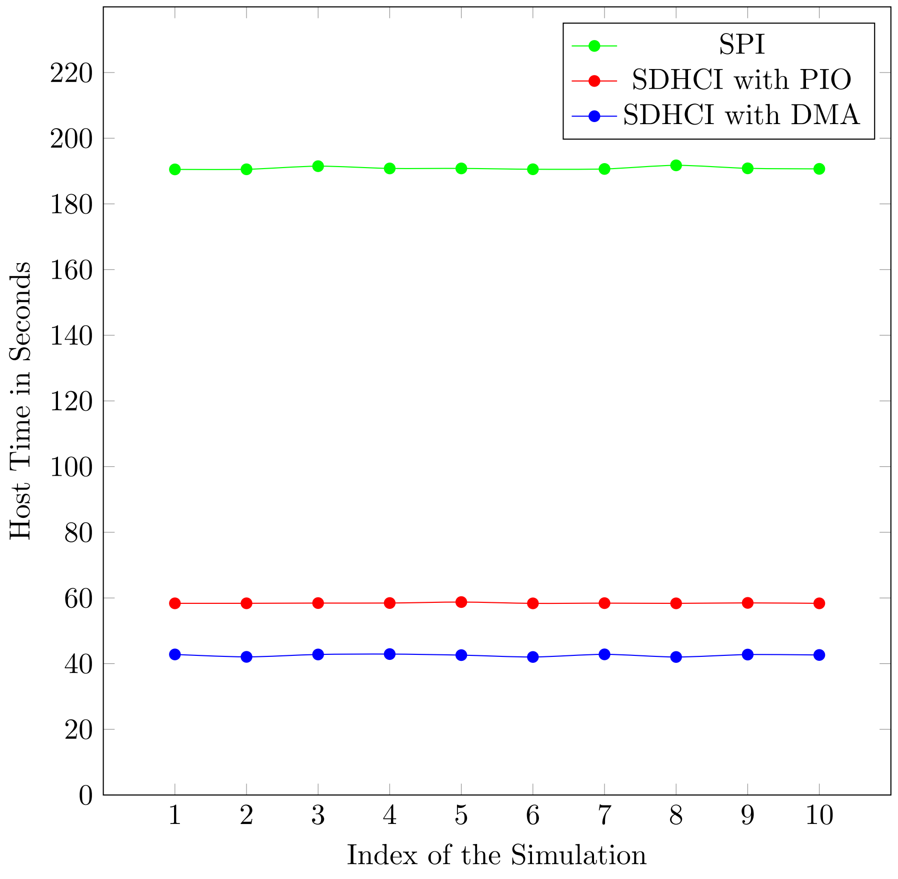


Figure 5.1: Results of the Speedup Evaluation

SDHCI with DMA	**SDHCI with PIO**	**SPI**
42.5469 sec	58.4422 sec	190.8553 sec

Table 5.1: Averages of the Speedup Evaluation

5.2 Speedup in the Linux Boot Time

A second benchmark for measuring the performance of the VP is the measurement of the boot time in total. Interesting is the real-time boot time - in the console output this is the runtime. This time is always displayed after the simulation of OR1kMVP has finished. The only question is how the simulation can be stopped automatically after the boot. To realize this, Jan Henrik Weinstock recently added the model `simdev.cpp` to the Virtual Components Modeling Library (VCML). It can be found in `src/vcml/models/generic` [9]. This device was connected to the system bus with the address 0x99500000. It has a *Stop* register at offset address 0. If this register is written, the function `sc_stop()` is called automatically and the whole simulation is stopped. This register must be written at the same point where the MD5 checksum was calculated in the previous section. Therefore, the command `devmem 0x99500000 w 1` was added to the init file. `Devmem` is a function that allows to read or write directly at a specified address in main memory. This command writes to the *Stop* register of the simdev device and thus terminates the simulation. Then, the boot time (runtime) can be read from the console output.

The improvement of the read and write performance described in section 5.1 of course directly affects a reduction of the boot time. The measurement results of the boot time can be seen in Figure 5.2 and Table 5.2. The Table shows that the boot time could be reduced from about 45.6 seconds in real time to about 17.3 seconds with PIO and 13.6 seconds with DMA. Altogether it results that boot via PIO transfer is $\frac{45.58155sec}{17.27436} = 2.64$ times as fast as the boot via SPI. The DMA boot is $\frac{17.27436sec}{13.55263sec} = 1.27$ times as fast as the boot via PIO. Thus, in total the boot via DMA is 3.4 times as fast as the boot via SPI.

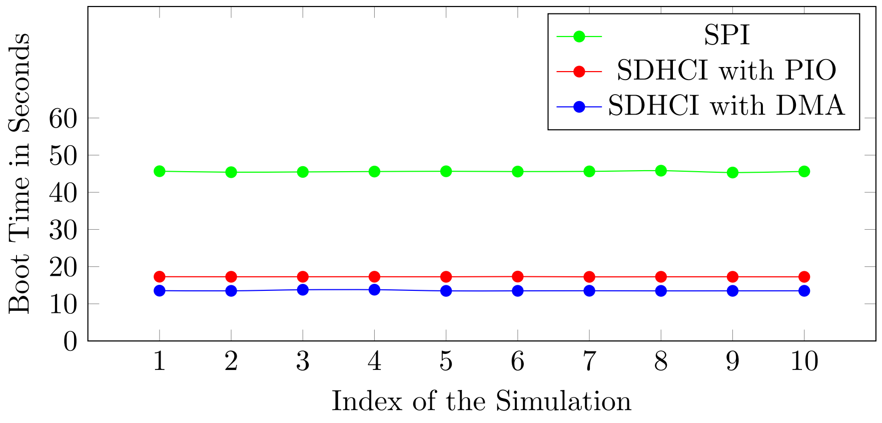


Figure 5.2: Results of the Boot Time Evaluation

SDHCI with DMA	**SDHCI with PIO**	**SPI**
13.5526 sec	17.2744 sec	45.5816 sec

Table 5.2: Averages of the Boot Time Evaluation

6 Conclusion and Future Work

In this Bachelor thesis a fully functional virtual model of an SD Host Controller Interface (SDHCI) was developed and tested. It has been written in SystemC TLM-2.0 and extends the Virtual Components Modeling Library (VCML) implemented by Jan Henrik Weinstock. The model is therefore compatible to any other Virtual Platform (VP) implemented in SystemC TLM. It can easily be instantiated from the VCML and bound to the virtual system bus. All information about the model and how to use it can be found in this thesis.

In the further course of the work Direct Memory Access (DMA) functionality for the SDHCI model was implemented. Thus, the communication between the system bus of the VP and an SD card could be accelerated up to the 4.5-fold compared to the communication via Serial Peripheral Interface (SPI).

In addition, Google Test unit tests for the SDHCI model were implemented to facilitate the correct integration of the model into a VP and the maintenance of the model.

It is the hope of the author that his SDHCI model as an Open Source Project will find many applications in research with Virtual Platforms and will speed up many VPs in the future.

The SDHCI model can be further developed in the future. Especially two extensions would be interesting. First, the SDHCI could be extended with a (dynamically) changeable number of virtual SD card slots. Then, for example, one SD card could contain the file system and another could serve as an additional memory. Both could be attached to the bus via the same SDHCI by getting different addresses on the SDHCI internal "SD bus". Thereby, a way needs to be found to enable the data flow and the use of the SDHCI registers for each SD card in parallel.
A second interesting extension of the model would be the introduction of SDIO. Therefore, an SDIO capable virtual SD card would have to be implemented first. This could be derived from the existing SD card written by Jan Henrik Weinstock and extended with the SDIO functionality. In the author's opinion, the SDHCI would only have to be modified in a few places, since SDIO commands are sent to the SD card just like normal SD commands.

Acronyms

ADMA Advanced DMA.

CPU Central Processing Unit.

DMA Direct Memory Access.

GDB GNU debugger.

OR1kISS OpenRISC 1000 Instruction Set Simulator.

OR1kMVP OpenRISC 1000 Multicore Virtual Platform.

OS operating system.

PIO Peripheral Input/Output.

SDHCI SD Host Controller Interface.

SDMA Single Operation DMA.

SPI Serial Peripheral Interface.

TLM Transaction Level Modeling.

VCML Virtual Components Modeling Library.

VP Virtual Platform.

List of Figures

List of Tables

List of Listings

References

[1] H. Herold, B. Lurz, and J. Wohlrad, *Grundlagen der Informatik*, 2nd ed. München, Germany: Pearson Deutschland GmbH, 2012.

[2] Linux Information Project, "MS-DOS: A brief Introduction," retrieved on July 11, 2019. [Online]. Available: http://www.linfo.org/ms-dos.html

[3] J. H. Weinstock, "OpenRISC 1000 Multicore Virtual Platform (OR1kMVP)." [Online]. Available: https://github.com/janweinstock/or1kmvp

[4] L. Urban, "Model of an SD Host Controller Interface for Virtual Platforms." [Online]. Available: https://github.com/lasseUrban/sdhci

[5] M. Igor and P. Crosthwaite, "SD Association Host Standard Specification v2.0 controller emulation." [Online]. Available: https://github.com/qemu/qemu/blob/master/hw/sd/sdhci.c

[6] Open Virtual Platforms, "OVP Peripheral Model: FreescaleVybridSDHC." [Online]. Available: http://www.ovpworld.org/library/wikka.php?wakka=FreescaleVybridSDHC

[7] T. de Schutter, *BetterSoftware. Faster!* Mountain View, CA, USA: Synopsis, Inc., 2014.

[8] *IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Computer Society, IEEE, 3 Park Avenue, New York, NY 10016-5997, USA, January 2012. [Online]. Available: https://paginas.fe.up.pt/~ee07166/lib/exe/fetch.php?media=1666-2011.pdf

[9] J. H. Weinstock, "Virtual Component Modelling Library (VCML)." [Online]. Available: https://github.com/janweinstock/vcml

[10] ——, "OpenRISC 1000 Instruction Set Simulator (OR1kISS)." [Online]. Available: https://github.com/janweinstock/or1kiss

[11] L. Torvalds, "Linux kernel." [Online]. Available: https://github.com/torvalds/linux

[12] eLinux, "Device Tree Usage." [Online]. Available: https://elinux.org/Device_Tree_Usage

[13] SD Association, "SD Specifications Part A2, SD Host Controller Simplified Specification," July 2018, version 4.20. [Online]. Available: https://www.sdcard.org/downloads/pls/index.html

[14] S. Association, "SD Standard Overview." [Online]. Available: https://www.sdcard.org/developers/overview/index.html

[15] A. Rubini and J. Corbet, *Linux Gerätetreiber*, 2nd ed. O'Reilly, 2002. [Online]. Available: https://www.oreilly.de/german/freebooks/linuxdrive2ger/memdma.html

[16] Google, "Googletest - Google Testing and Mocking Framework ." [Online]. Available: https://github.com/google/googletest

[17] SD Association, "SD Specifications Part 1, Physical Layer Simplified Specification," August 2018, version 6.00. [Online]. Available: https://www.sdcard.org/downloads/pls/index.html