Michael Moritsch, BSc

# Verification of a combined passive HF/UHF RFID Tag with Universal Verification Methodology

**Master Thesis**

**MA 719**

Graz University of Technology

Institute of Electronics
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Bösch

Supervisor: Ass.-Prof. Dipl.-Ing. Dr.techn. Peter Söser
External Supervisor: Dipl.-Ing. Johannes Schweighofer

Graz, August 2012

# Abstract

For every digital design verification is very important. It takes a major part of the effort and additionally the costs to fix a bug grows exponentially the further the product gets in the process. Therefore it is important to find failures as early as possible, best in the RTL-simulation. The Comprehensive Transponder System (CTS) called RFID platform needs a new Test Bench which allows exhaustive constraint-random driven verification. The Universal Verification Methodology (UVM) is used to implement the required verification environment. UVM is a methodology that allows the development of reusable constraint-random driven Test Benches. Basically it is a class library implemented in SystemVerilog, a enhancement to Verilog. The result is a Test Bench that allows the automatic verification of the CTS over several interfaces. The stimuli are generated randomly and functional coverage is collected to control what has been verified. The new Test Bench allows a more complete and easier verification of the device under test.

# Kurzfassung

Für jedes digitale Design ist die Verifikation von großer Bedeutung. Sie benötigt einen Großteil des Aufwandes und außerdem steigen die Kosten um einen Fehler auszubessern exponentiell umso weiter das Produkt in seiner Entwicklung fortgeschritten ist. Deshalb ist es wichtig, Fehler so früh wie möglich zu erkennen, am besten schon in der RTL-Simulation. Die Comprehensive Transponder System (CTS) genannte RFID Plattform benötigt eine neue Test Bench, die eine eingehende Überprüfung mit eingeschränkt zufälligen Test Vektoren ermöglicht. Die Universal Verification Methodology (UVM) wurde benutzt um die erforderliche Verifikationsumgebung zu implementieren. UVM ist eine Methodik, die die Entwicklung einer wiederverwendbaren Test Bench ermöglicht. Grundsätzlich handelt es sich dabei um eine Klassen-Bibliothek, welche in SystemVerilog implementiert ist. SystemVerilog ist eine Erweiterung von Verilog. Das Ergebnis ist eine Test Bench die die automatische Überprüfung des CTS über mehrere Schnittstellen erlaubt. Die Testvektoren werden zufällig generiert und zusätzlich wird die "functional coverage" gesammelt. Diese ermöglicht es zu kontrollieren, was schon überprüft wurde. In Summe ermöglicht die neue Test-Umgebung eine vollständigere und einfachere Verifikation des Prüfobjektes.

## Statutory Declaration

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

Graz, _____ _____

Date Signature

## Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

# Acknowledgement

First of all I would like to thank Ass.-Prof. Dipl.-Ing. Dr.techn. Peter Söser, member of Institute of Electronics at the Graz University of Technology, for supervising my thesis.

I would also like to thank Dipl.-Ing. Gerald Hohlweg, head of the Contactless and Radio Frequency Exploration department (CRE) of Infineon Technologies Austria AG, for the opportunity to write my thesis in a practical environment.

Special thanks to my supervisor at Infineon Austria in Graz, Dipl.-Ing. Johannes Schweighofer for his great support.

Furthermore, I would like to thank my colleagues at CRE for the positive and creative work atmosphere and for plenty of amusing moments.

Last but not least I would like to express my gratitude to my parents Margit and Peter Moritsch and my sister Petra for supporting me during my whole life.

# Contents

# List of Figures

# List of Tables

# List of Listings

# Acronyms

**BLF** Backscatter Link Frequency

**CDRV** coverage driven constraint random verification

**CPHA** Clock Phase

**CPOL** Clock Polarity

**CTS** Comprehensive Transponder System

**DUT** Device Under Test

**EEPROM** Electrically Erasable Programmable Read-Only Memory

**EPC** Electronic Product Code

**HF** High Frequency

**NFC** Near Field Communication

**NVM** Non Volatile Memory

**PC** Protocol-control

**PIE** Pulse Interval Encoding

**RFID** Radio-frequency identification

**RFU** Reserved for Future Use

**RTL** Register-transfer level

**SPI** Serial Peripheral Interface Bus

**TLM** Transaction-Level Modeling

**UHF** Ultra High Frequency

**UVM** Universal Verification Methodology

**XPC** Extended Protocol Control

# 1 Introduction

Verification is a fundamental part of every digital design. Usually 60% to 80% of design effort goes to verification. Besides, the longer a bug is undetected, the more expensive it is. To fix a bug during a simulation phase, will not cost a lot. But fixing a bug after product release can cost millions of dollars to fix. Not to mention the loss of reputation.

For the verification of a Register-transfer level (RTL) implementation usually a Test Bench is used. A "Test Bench mimic the environment in which the design will reside. It checks whether the RTL Implementation meets the design spec or not. This Environment creates invalid and unexpected as well as valid and expected conditions to test the design" [4].

The simplest and fastest way is to write a linear Test Bench. Typically this type of Test Bench is written in VHDL or Verilog. The stimuli (test vectors) are often hard coded and the results are checked manually. This approach may be appropriate for small designs or for a first verification. But it is not adequate for large designs. It is only possible to hit some corner cases and of course it creates a large effort to control the results.

The better way is to use coverage driven constraint random verification (CDRV) architectures. Constraint random means that the test vectors are not just randomly generated. The randomization is constrained in a way to only generate exploitable values. Because every point in the test plan is generated automatically, a mechanism is needed to tell which points are verified. This mechanism is functional coverage. Both, VHDL and Verilog are not really applicable to write such a Test Bench. Therefore the Accellera Systems Initiative developed SystemVerilog. This enhancement of Verilog offers several constructs to allow the easy development of a CDRV Test Bench.

Another important point is to develop the Test Bench to be reusable. Especially when testing large designs this can help a lot to save implementation effort. The Universal Verification Methodology (UVM) is a methodology to design coverage driven constraint random Test Benches which consist of reusable components. In fact UVM is implemented as a SystemVerilog class library.

The aim of this thesis is to develop an UVM based CDRV Test Bench for a Comprehensive Transponder System (CTS). Basically the system is a passive Radio-frequency identification (RFID) tag which supports High Frequency (HF), Ultra High Frequency (UHF) and also Near Field Communication (NFC) communication. The developed Test Bench should verify the HF/UHF interfaces as well as an also available Serial Peripheral Interface (SPI) interface. The requirements are that the verification environment should be modular and reusable as well as easily expandable. Furthermore it should be configurable and constraint random driven.

## 1.1 Chapter Overview

Chapter 2 outlines Radio-frequency identification (RFID) systems and also describes the Electronic Product Code (EPC) standard.

In Chapter 3 the Serial Peripheral Interface (SPI) bus is described.

Chapter 4 introduces SystemVerilog and describes new features of this language.

In Chapter 5 of the basics of Universal Verification Methodology (UVM) which is used to develop the Test Bench are described

Chapter 6 gives a short overview over the Device Under Test (DUT).

Chapter 7 demonstrates the developed Test Bench.

Finally Chapter 8 forms the conclusion of this thesis.

# 2 RFID Systems

Radio-frequency identification (RFID) describes a technique which uses electromagnetic waves to identify an object. A so called RFID tag basically consists of an antenna, an analog module, a digital logic and a persistent memory. Tags can be distinguished into passive and active tags. Where active tags have their own power source, passive ones receives the whole energy from the interrogator/reader (Figure 2.1). The majority of the tags are passive ones.

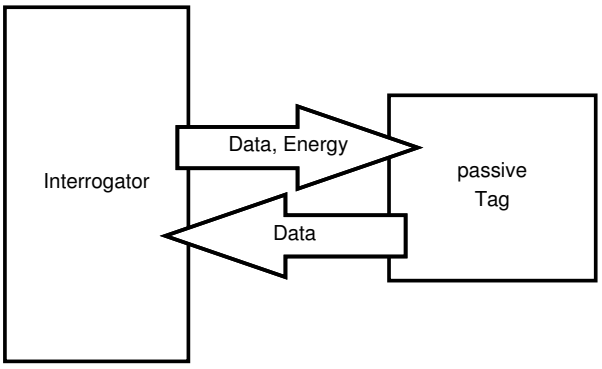


Figure 2.1: RFID-System with a passive tag

RFID systems works in many different frequency ranges and there are various different protocols, some are standardized others are not.

The following sections will focus on the two standards which where used in this a work. They are developed by an organization called EPCglobal and defines a standard for the 13.56 MHz (HF) and UHF (860 - 960 MHz) frequency bands.

## 2.1 EPC Class-1 Generation-2 UHF

This specification [11] defines the communication for a passive RFID system at a frequency range of 860 MHz to 960 MHz. It specifies the analog requirements as well as the digital ones. The tags are passive which means that the whole energy is supplied by the interrogator over electromagnetic wave propagation.

### 2.1.1 Interrogator-to-Tag communications

The interrogator transmits data to the tag by modulating the RF carrier. Allowed modulation techniques are double-sideband amplitude shift keying (DSB-ASK), single-sideband amplitude shift keying (SSB-ASK), or phase-reversal amplitude shift keying (PR-ASK). The used encoding technique is Pulse Interval Encoding (PIE), shown

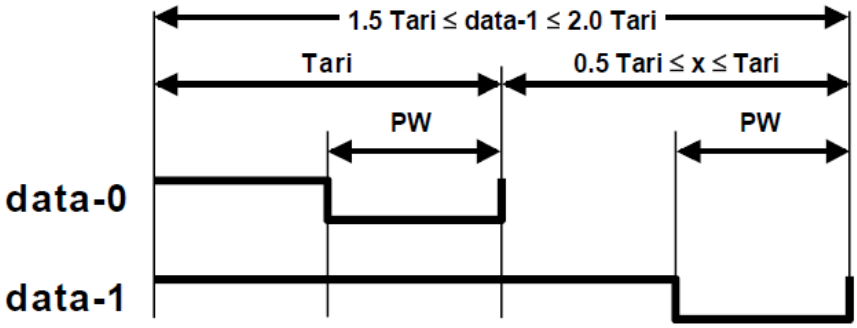


Figure 2.2: PIE encoding used by the interrogator to communicate with the tag. [11]

in Figure 2.2. The allowed range for the reference time Tari is 6.25 $\mu s$ to 25 $\mu s$. Before each command either a preamble or a frame-sync, shown in Figure 2.3, is send. The

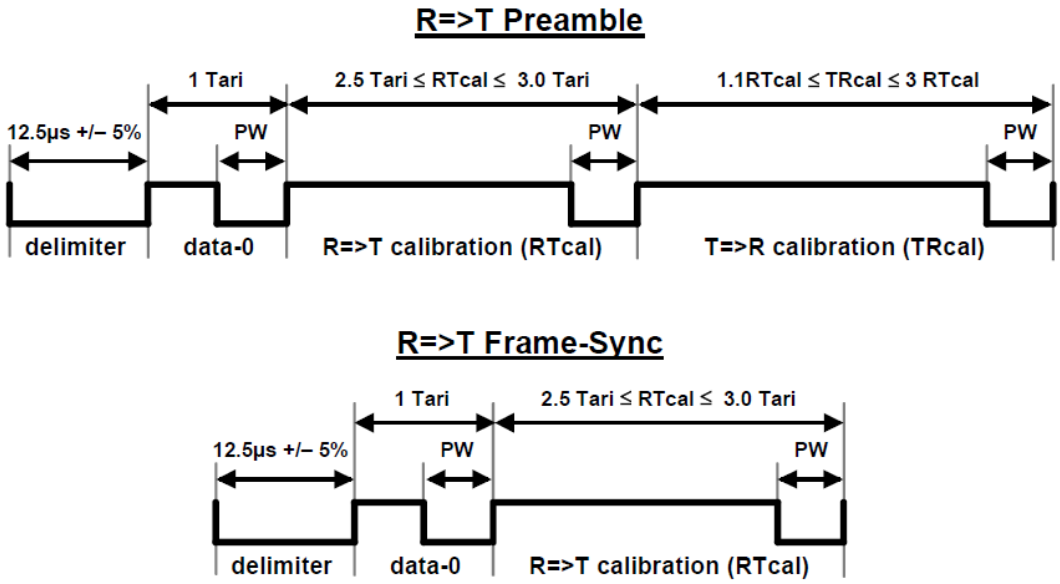


Figure 2.3: Preamble and Frame-Sync which are send before each command. [11]

RTcal calibration symbol has the length of data-0 plus the length of a data-1. The tag

calculates a $pivot = RTcal/2$ and uses it to decide if a symbol is a data-0 (smaller than pivot) or a data-1 (larger than pivot). The TRcal is used to specify the tag's Backscatter Link Frequency (BLF).

$$BLF = \frac{DR}{TRcal} \tag{2.1}$$

Where $DR$ stands for divide ratio and is defined by the interrogator in the QUERY command. The allowed values are $64/3$ and 8.

### 2.1.2 Tag-to-Interrogator communications

The tag answers by backscatter modulating the amplitude and/or the phase of the unmodulated RF carrier. This is done by switching the reflection coefficient of the tag's antenna between two states. The data encoding is chosen by the interrogator (in the QUERY command) and can be either FMo or Miller encoding. In case of Miller encoding two, four or eight sub-carrier cycles per bit are possible. Independent of the encoding the transmission always ends with a dummy-1 symbol. If specified in the QUERY, before the preamble a pilot tone is sent.

**FMo encoding** does have a baseband phase inversion at every symbol border. A data-0 symbol has an additional inversion at the middle of the symbol. Figure 2.4 shows symbol sequences for two bit combinations. FMo encoding is chosen by setting the M value zero.

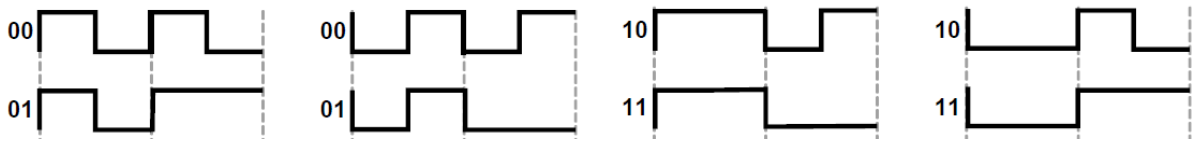


Figure 2.4: FMo Sequences [11]

**Miller encoding** inserts a baseband phase inversion between two consecutive data-0 symbols. Data-1 symbols have an inversion in the middle of the symbol. Depending on the M value, a Miller sequence has exactly two, four or eight sub-carrier cycles per symbol. Figure 2.5 demonstrates some symbol sequences for different values of M.

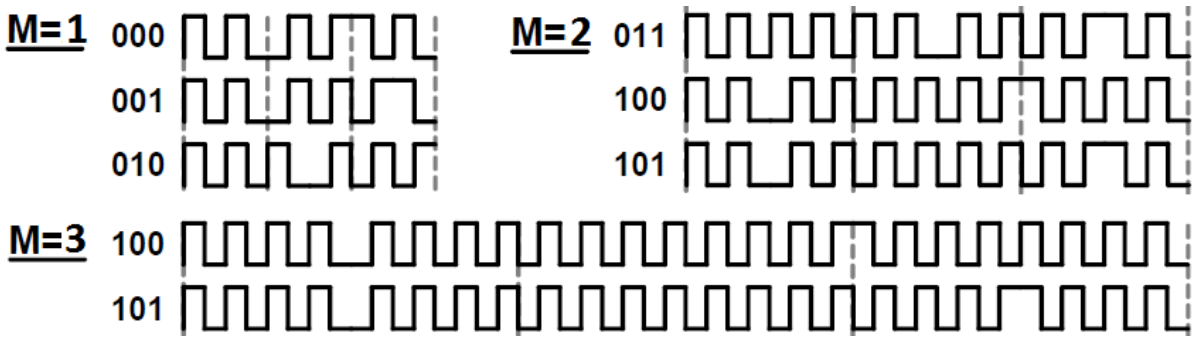


Figure 2.5: Miller Sequences [11]

### 2.1.3 Tag memory

The memory is logically divided into four banks (Figure 2.6). Each bank contains zero or more 16-bit words. Memory manipulating commands operate in word boundaries. The memory banks are:

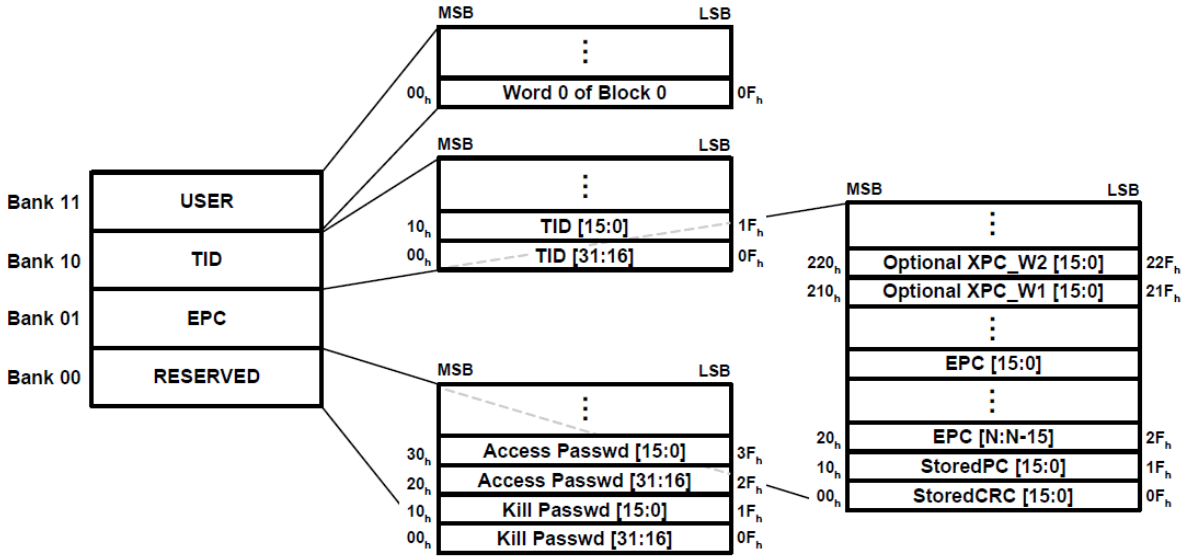


Figure 2.6: Tag memory banks and logical addresses. [11]

- Reserved: contains the kill password and the access password, both are 32-bit long.
- EPC: contains a StoredCRC, a Protocol-control (PC) word and the Electronic Product Code (EPC). The PC contains information about, for example, the

length of the EPC or if the tag has a User-memory. The StoredCRC is calculated over the PC and the EPC at power-up. Tags which have the Extended Protocol Control (XPC) implemented also store two additional words.

- TID: contains an 8-bit ISO/IEC 15963 allocation class identifier and additional information over custom commands and/or optional features that a tag supports [11].
- User: is optional and can contain whatever a user wishes.

### 2.1.4 Link timing

The communication between interrogator and tag has to fulfill several timing requirements, shown in Figure 2.7. The parameters defines the following times:

- $T_1$ defines the time inside which a response is valid. It is the time from the last rising edge of the last bit of the interrogator transmission to the first rising edge of the tag response [11]. A tag may exceed $T_1$ in case of a memory access (for example at a WRITE command the answer has to be inside 20 $\mu s$).
- $T_2$ defines the minimum time interval between a tag's response and a new command sent by the interrogator.
- $T_3$ is the time an interrogator waits, after $T_1$ expired, before sending a new command.
- $T_4$ defines the minimum time between two commands.

Table 2.1 shows the minimum, nominal and maximum values for each parameter. Where $T_{pri}$ is the period of the response and calculated as:

$$T_{pri} = \frac{1}{BLF} = \frac{TRcal}{DR} \tag{2.2}$$

$FT$ is the frequency tolerance, for the specification see [11, table 6.19].

| Parameter | Minimum | Nominal | Maximum |
|---|---|---|---|
| $T_1$ | $max(RTcal, 10T_{pri})$ $(1 - \lvert FT \rvert) - 2\mu s$ | $max(RTcal, 10T_{pri})$ | $max(RTcal, 10T_{pri})$ $(1 + \lvert FT \rvert) + 2\mu s$ |
| $T_2$ | $3.0T_{pri}$ | | $20.0T_{pri}$ |
| $T_3$ | $0.0T_{pri}$ | | |
| $T_4$ | $2.0T_{pri}$ | | |

Table 2.1: Link timing parameters [11]

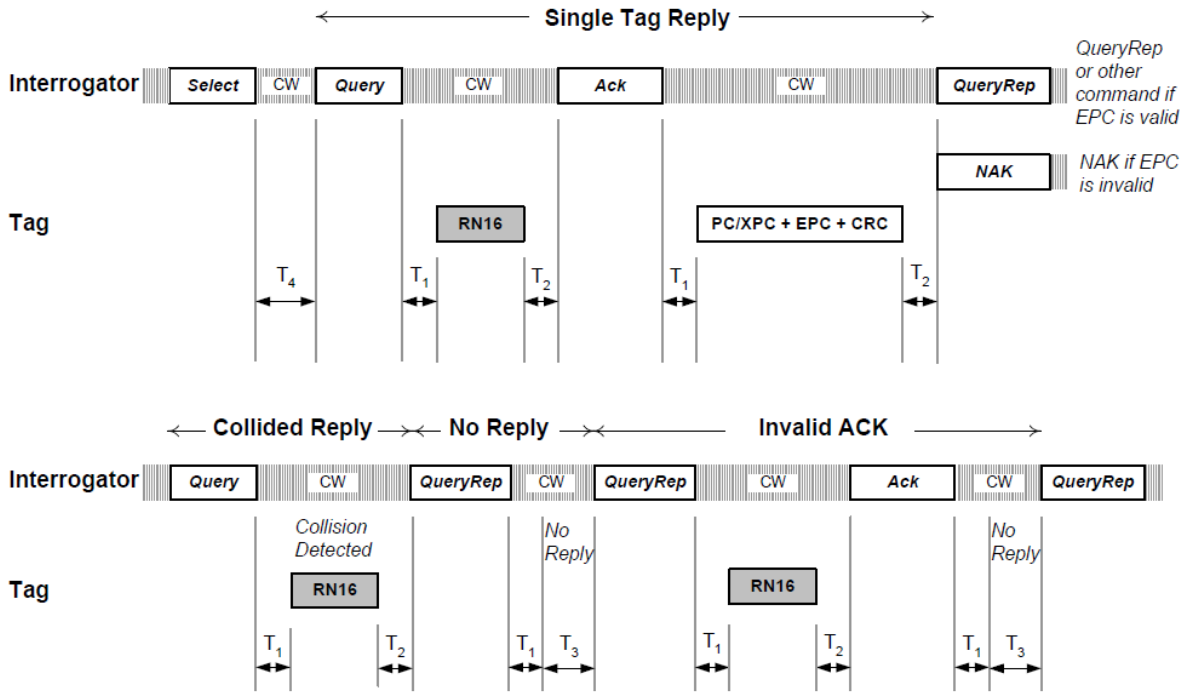


Figure 2.7: Link timing [11]

### 2.1.5 Communication procedure

To be able to manage tag populations every tag has a selected flag (SL) and can be inventoried in one of four sessions. For each of these sessions an inventoried flag exists and can either be A or B. These flags can be manipulated with the SELECT command.

To avoid collisions a singulation method is implemented. For this purpose the tag has a 15-bit slot counter which is preloaded with a random value between 0 and $2^Q - 1$. Q has a range from 0 to 15 and is initially set by the QUERY command. Anyway, the slot counter can also be manipulated with the help of the QUERY_ADJUST as well as the QUERY_REP command. The first one increases or decreases the Q value and the tag generates a new random slot counter value afterwards. The QUERY_REP command simply decreases the slot counter value by one.

Every inventory round starts with a QUERY command. The command specifies a session, a proper inventoried flag value and a selected flag value. Tags with mismatching flags, ignore the command and stay in READY state. If the flags are matching but the slot counter is not equal to zero the tag changes to the ARBITRATE state. At this moment the slot counter becomes zero, manipulated before by any Query command, the tag responds with a 16-bit random number (RN16) and the state changes to REPLY.

After receiving an ACK command, with the same RN16, the tag changes to ACKNOWLEDGED and replies the PC word, the stored EPC and the stored CRC word.

The follow-up command is a REQ_RN. Whether the RN16 is valid or not, the tag answers with a new 16-bit random number (denoted handle) and changes the state. If the, on the tag stored, access password is not equal to zero the new state is OPEN, else the tag changes directly to SECURED. Consecutive commands uses the handle as an identification parameter.

Tags in the OPEN state are able to execute memory manipulating commands like READ and WRITE as well as the KILL command. Additionally to this commands in the SECURED state the tag can execute the LOCK command which allows the user to lock/unlock parts of the memory. To reach the SECURED state the ACCESS command is used. At last killed tags are in the KILLED state in which they are not answering to anything. A killed tag stays killed forever.

Figure 2.8 shows all possible states a tag can be in, as well as the permitted transactions between them.

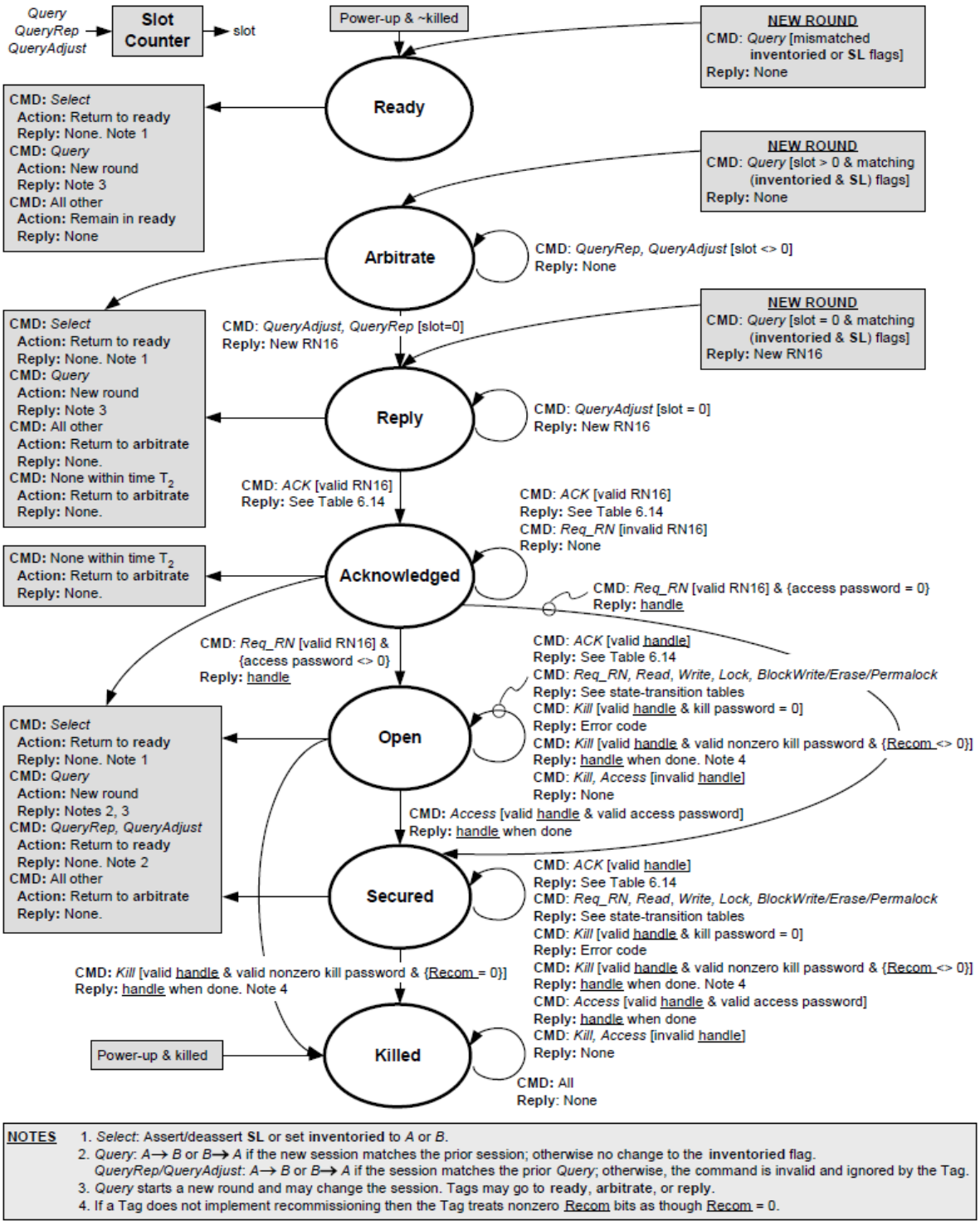


Figure 2.8: Tag state diagram. [11]

### 2.1.6 Commands

This subsection gives a brief overview of all (mandatory) commands of the EPC standard. Basically only the fields of the commands and, if there is one, of the responses are explained.

**Select** is used to manipulate the SL and the inventoried flags of a tag. The command has a Target field to define which flag should be modified. The Action field defines how the flag is changed. With the fields MemBank, Pointer and Length a part of the memory is selected. Only if this part matches with the field Mask the tag performs the action. The Truncate bit specifies if the tag should truncate the response on a ACK. Lastly the command is secured by a 16-bit long CRC field.

**Query** initiates and specifies an inventory round. The DR field defines the TRcal divide ratio which is used to calculate the BLF. M affects the kind of response coding technique. If a response contains a pilot tone or not, is defined by TRext. The fields Sel, Session and Target specify which tags are effected by this command. Q is used by the tag to calculate the initial value of the slot counter. A 5-bit CRC field secures the command. A Query may respond with a 16-bit long random number (RN16).

**QueryAdjust:** Dependent on the UpDn field the tag increases or decreases the Q value by 1, but only at tags which match the Session specified in the command. There is no CRC field, but a 16-bit random number as response is possible.

**QueryRep** decreases the slot counter by 1 at tags with the appropriate session. The command is not secured and may respond a random number.

**ACK:** If the random number in the RN field is valid (RN16 in Reply or Acknowledge state, Handle in Open or Secured state), the command respond with [PC,EPC,CRC-16].

**NAK:** After receiving the tag returns to ARBITRATE state, except it is currently in ready or killed state.

**Req_RN** returns the handle or a new 16-bit random number. The RN field either contains the RN16 from a Query or the handle. The command and the response are secured by a CRC-16.

**Read** is used to read one or more words from the memory location specified by the fields MemBank, WordPtr and WordCount. The field RN must contain the valid handle otherwise the command is ignored. If the location is valid the tag responds with a one bit header set to 0, the desired data and the handle. Else the header bit is 1 and an error code is returned. Both command and response are secured by a CRC-16 field.

**Write** allows to write a word into the memory on a location specified by MemBank and WortPtr. The remaining fields are Data and RN which contains the handle. Data is xored with a 16-bit random number that is calculated by a prior Req_RN. In case of a successful operation the response is [0, handle, CRC-16], else an error code is returned.

**Kill:** To kill a tag, two consecutive kill commands are necessary. Each of them contains half of the 32-bit kill password. Additionally there is an RN field with the handle and a CRC-16 field. The response on the first kill is the handle and a CRC-16. If the second kill was successful the response is [0, handle, CRC-16]. The kill command also has an RFU/Recom field which allows to recommission the tag instead of killing it.

**Access:** The 32-bit access password is send to the tag via two consecutive commands. Additionally there is the RN and the CRC field. In case of success the tag replies with the handle secured by a 16 bit CRC.

**Lock** allows to lock/unlock or permanent lock the kill or access password or one of the other memory banks. The passwords will not be writeable and not be readable if locked. The banks will only be write locked. The Payload field contains the information which memory is affected and how.

## 2.2 EPC Class-1 HF

This specification [12] defines the communication for a passive RFID system at a frequency range of 13.56 MHz. The main difference to the UHF standard can be found in the analog area. In this case energy is transmitted over inductive coupling and the tag to interrogator communication is done with load modulation. This means that a tag varies its "load" to respond to the reader (for example by switching load resistors or shunt diodes).

From a protocol point of view there are only minor differences. Additionally to FM0 and Miller encoding, the standard defines Manchester encoding as another possibility for the tag-to-interrogator link. An M value equal to zero means FM0 encoding, one means Miller encoding with eight sub-carriers and two and three means Manchester encoding. Instead of using a dummy-1 to terminate a transmission an EOF symbol is used. Figure 2.9 shows the EOF for a FM0 transmission.

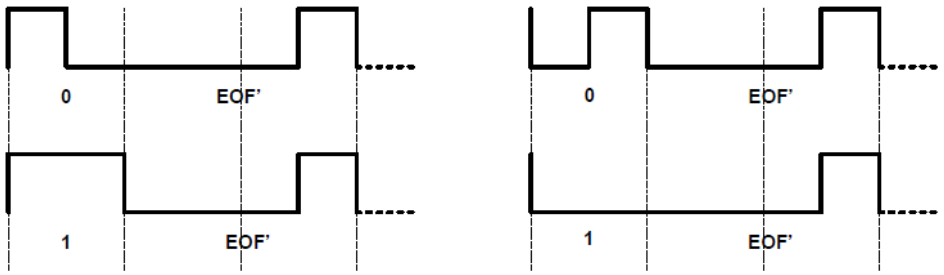


Figure 2.9: Terminating FM0 transmissions EOF [11]

**Manchester encoding** uses transitions in the middle of a symbol to encode bits. A high to low transition represents a data-0 and a low to high transition a data-1. A Manchester symbol sequence should have four or eight sub-carrier cycles per symbol, depending on M. Figure 2.10 presents Manchester encoded sequences for M equal to two and three.

As well only two link frequencies exist, 424 $kHz$ and 847 $kHz$. Besides that, the link timing parameters are different as shown in Table 2.2.

Anyway, also the names of the QUERY, QUERY_ADJUST and QUERY_REP commands changed to BEGIN_ROUND, RESIZE_ROUND and NEXT_SLOT, as well their response now has an additional CRC field.

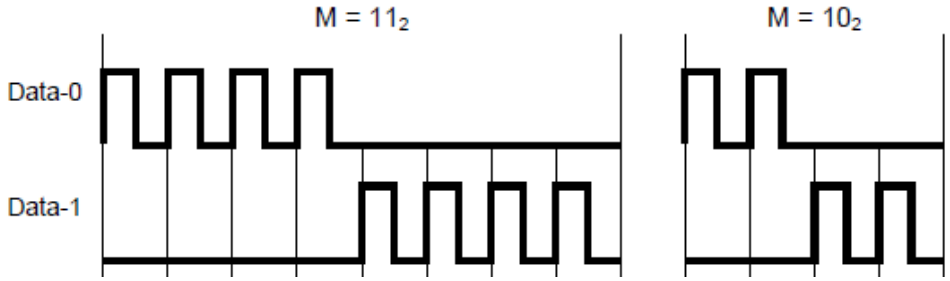


Figure 2.10: Manchester Sequences [12]

| Parameter | Minimum | Nominal | Maximum |
|---|---|---|---|
| $T_1$ | 73.1 $\mu s$ | 75.5 $\mu s$ | 77.9 $\mu s$ |
| $T_2$ | 151 $\mu s$ | | 1208 $\mu s$ |
| $T_3$ | $T_{sof\_tag}$ | | |
| $T_4$ | $T_{1Typ} + T_{3Min}$ | | |

Table 2.2: HF link timing parameters [12]

# 3 Serial Peripheral Interface Bus (SPI)

The Serial Peripheral Interface (SPI) describes a synchronous serial bus interface, which is used to connect digital devices in a master/slave principle. The standard, named by Motorola, specifies only a few things and gives the user many possibilities for customization.

## 3.1 Specifications

Figure 3.1 shows a single master and single slave configuration. Also single master, multi slave configurations are possible.

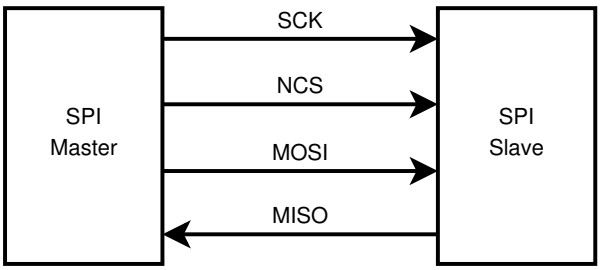


Figure 3.1: SPI in a single master, single slave configuration

The four specified logic signals have the following purpose:

- SCK: SPI data clock
- NCS: SPI chip select (active low)
- MOSI: Master out, slave in data signal
- MISO: Master in, slave out data signal

The signal names are not specified, therefore alternative naming conventions are also widely used [16].

The major specification is that read and write operations have to be on different edges (for example, read on the positive edge and write on the negative one). On which edge which operation is done depends on the values of the Clock Polarity (CPOL) and Clock Phase (CPHA) options. These parameters have to be defined by the master before a transaction. CPOL defines the base (idle) value of the clock, therefore $CPOL = 0$ means the clock is idle low, similarly if $CPOL = 1$ the clock is idle high. CPHA defines if the data is sampled on the first ($CPHA = 0$) or on the second edge after NCS has gone low. Therefore for example if $CPOL = CPHA = 0$, data is read on the positive edge and written on the negative. To describe which configurations are supported, often one of the modes specified in Table 3.1 are used.

| Mode | CPOL | CPHA |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 0 | 1 |
| 3 | 1 | 0 |
| 4 | 1 | 1 |

Table 3.1: SPI modes [16]

The SPI specification does not define a specific data format or a protocol which has to be used. Wherefore these things are user specific. In the following the specifications used by the SPI interface of the Device Under Test (DUT) are presented.

## 3.2 Data Format

A SPI frame consists of several 16-bit words and the bits are ordered MSB first. There are four different words:

- The **instruction word** consists of a 2-bit command field (bit 15 and bit 14), a Reserved for Future Use (RFU) field (bits 13 to 4) and an optional Non Volatile Memory (NVM) operation field (bits 3 to 0).
- The **address word** consists of a 6-bit RFU field (bits 15 to 10) and a 10-bit address field (bits 9 to 0).
- The **data word** contains the 16-bit data word. This would either be the data to read or to write.

| SPI Command | Description |
|---|---|
| 00 | RFU |
| 01 | Write data |
| 10 | Read data |
| 11 | Get status |

Table 3.2: SPI instructions

The RFU fields are reserved for future use and have to be kept low. The NVM field in the instruction word refers to the various access modes of the NVM and is optional. If it is not needed it has to be kept low and the default read or write modes would be used. Table 3.2 shows the possible values for the 2-bit command field of the instruction word.

## 3.3 Transfer Modes

Every transmission starts at the moment the master sets NCS to low. Similarly it ends if NCS is reset back to high. It is possible to send multiple frames during the time the chip is selected. By simple changing the instruction it is possible to alter the mode while the device is selected. There are three possible modes.

**Write to the device:** Figure 3.2 shows two successive write frames (block write). First the master selects the device by setting NCS low and afterwards the instruction, address and write data (WDATA) words are shifted in on MOSI. An inactive (idle) time between the last bit of WDATA and the first bit of a consecutive instruction is mandatory in case of data written to the NVM. As it can be seen at the second write operation an idle phase between two words is allowed. During this time the clock has to stay in its idle state. To complete the write operations the master sets NCS to high.

**Read from the device:** Figure 3.3 demonstrates two consecutive read frames (block read). The master selects the device by setting the NCS line low. After this, the instruction and address words are shifted in on MOSI. The device fetches the data from the address, which afterwards is shifted out on MISO. The time the device needs to fetch the data depends on the location. The master has to wait the appropriate time and must keep the clock idle during waiting.

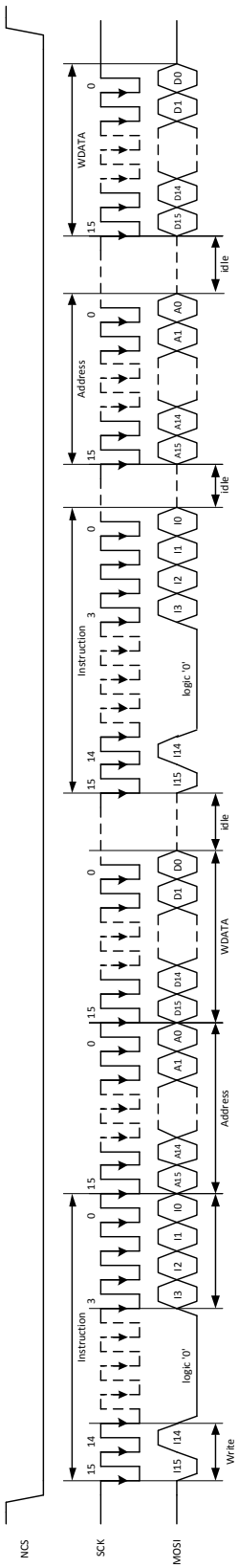


Figure 3.2: Block Write Transfer [14]

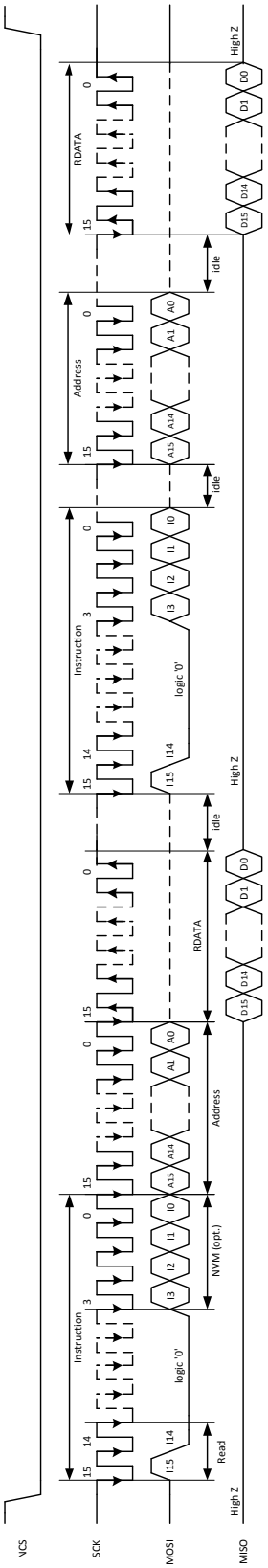


Figure 3.3: Block Read Transfer [14]

**Read device status** The last mode offers the possibility to read out the device status. Figure 3.4 shows such an operation. After NCS goes low the instruction word is shifted in on MOSI. At the moment the slave receives the last bit it starts to shift out the status word followed by the address word and the data word used by the last read or write operation. If the last instruction was a read, the data word would be $0x0000$. The status word consists of an RFU field (bits 15 to 7) and an address error status bit (bit 6) which informs if the last operation has produced an address error. Further it holds the SPI instruction field (bits 5 and 4) and the NVM operation field (bits 3 to 0) of the last read or write operation.

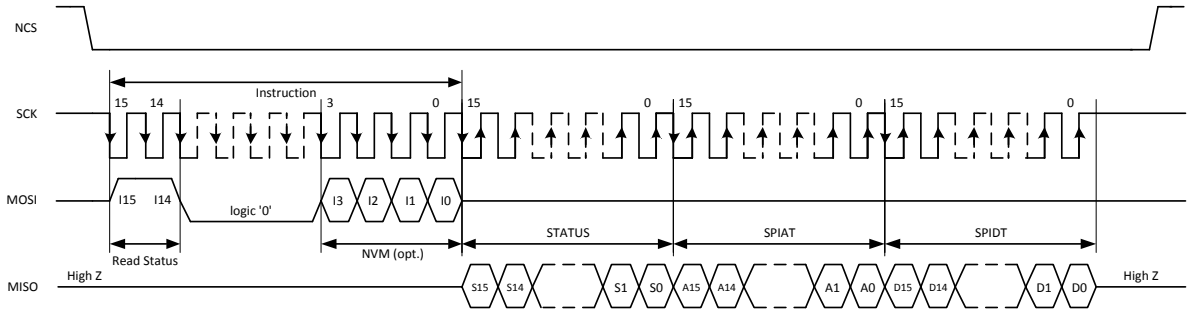


Figure 3.4: Read SPI Status [14]

# 4 SystemVerilog

SystemVerilog is the first hardware description and verification language (HDVL) and is a superset of Verilog-2005. It was developed by the Accellera Systems Initiative which is an "independent, not-for profit organization dedicated to create, support, promote, and advance system-level design, modeling, and verification standards for use by the worldwide electronics industry" [1].

SystemVerilog can be divided into two parts, one for modelling hardware designs on Register-transfer level (RTL) and another for verification. The RTL part is an enhancement of Verilog-2005. The verification part has much more similarities to software programming languages like C++ or Java then to Verilog. It uses extensive object-oriented programming techniques and is generally not synthesizable. The remainder of this chapter is mostly based on [5] and describes the new features of SystemVerilog. Good SystemVerilog references can also be found under [3] and [4].

## 4.1 New data types

SystemVerilog additionally offers several new integer data types. This new types, as well as the known types from Verilog, are shown in Table 4.1. In addition to $'0'$

| | |
|---|---|
| `shortint` | 2-state SystemVerilog data type, 16 bit signed integer |
| `int` | 2-state SystemVerilog data type, 32 bit signed integer |
| `longint` | 2-state SystemVerilog data type, 64 bit signed integer |
| `byte` | 2-state SystemVerilog data type, 8 bit signed integer or ASCII character |
| `bit` | 2-state SystemVerilog data type, user-defined vector size |
| `logic` | 4-state SystemVerilog data type, user-defined vector size |
| `reg` | 4-state Verilog-2001 data type, user-defined vector size |
| `integer` | 4-state Verilog-2001 data type, 32 bit signed integer |
| `time` | 4-state Verilog-2001 data type, 64 bit unsigned integer |

Table 4.1: Integer data types [5]

and $'1'$, 4-state types can also have unknown (X) and high-impedance (Z) values.

As an advantage, 2-state types may take less memory and simulates faster. When converting a 4-state to a 2-state type, the values X and Z will be handled as '0'.

Besides the `real` data type, which is similar to `double` in C, SystemVerilog also offers the data type `shortreal`, similar to the C data type `float`. The `void` data type allows functions without a return value.

SystemVerilog also introduces a `string` data type, which internally is represented as a dynamic-allocated byte array. Several operations on strings are possible, for example comparison, concatenation or indexing. There are also a number of special methods available. For instance the method `str.len()` returns the number of characters in the string `str`. For further details see [5] chapter 3.7.

## 4.2 Arrays

### 4.2.1 Packed and unpacked arrays

SystemVerilog makes a differentiation between "packed arrays" and "unpacked arrays". The first one refers to dimensions declared before the object name, the second to dimensions declared after the object name (see Listing 4.1 line 1,2).

A packed array is treated as an integer with a user-defined size, with the additionally advantage to be able to directly access every subfield (e.g. every bit). Such arrays can only be made of single bit data types (e.g. bit, logic, reg, ...). SystemVerilog also allows multidimensional packed arrays.

Unpacked arrays can be made of all data types and also of unpacked arrays. It is possible to create multidimensional arrays. To specify the size of an unpacked array, SystemVerilog also accepts a single number instead of a range (Listing 4.1 line 4,5).

```
1  bit [7:0] a; //packed array
2  int b[7:0];    //unpacked array
3
4  integer c[0:9][0:4]
5  integer d[10][5];  //same as c
6
7  //allowed operations on packed and unpacked arrays
8  //A and B has to be arrays of the same shape and type
9  A = B;  //reading and writing the array
10 A[i] = B[j]   //read/write an element of the array
11 A[i:j] = B[i:j];  //read/write a slice of the array
12 A[x+:c] = B[y+:c];  //read/write a variable slice of the array
13 A==B, A[i:j] != B[i:j]; //equality operations
14
15 //allowed operations on packed arrays only
```

```
16 A = 10'b11111111;  //assignment from an integer
17 A = A + 3;         //treatment as an integer
```

Listing 4.1: Array examples [5]

The lines 9 to 13 of Listing 4.1 shows which operations can be performed on packed and on unpacked arrays, whereas the operations at the lines 16 and 17 are only permitted on packed arrays. Assignments on unpacked arrays are done by assigning each element of the source array to its corresponding element of the target array. Packed arrays are treated like a vector, therefore every vector expression can be assigned to them.

### 4.2.2 Dynamic arrays

Dynamic arrays are unpacked arrays which size is not known at compile time. Their size can be set or changed at runtime. It should be mentioned that dynamic arrays are always indexed from 0 to `size-1`.

As it can be seen in Listing 4.2, dynamic arrays are declared by leaving the square brackets empty. The `new[]` operation is used to allocate or reallocate memory for the array. The `delete()` method empties the array, resulting an empty (zero-sized) array. The current size of an array can be discovered with the help of the `size()` method.

```
1  integer a[];   //zero-sized dynamic array of integers
2
3  //declare a dynamic array of 10-bit vectors and create 100 elements
4  bit [10:0] b[] = new[100];
5
6  //double the array size, preserving previous values
7  b = new[200](b);
8
9  int s = a1.size; //save the size of a1 into s
10 b = new[s * 4](b); //quadruple a1 array
11
12 b.delete;  //delete the array contents
13 $display("%d", b.size);  //prints 0
```

Listing 4.2: Dynamic arrays examples [5]

### 4.2.3 Associative arrays

Associative arrays can be seen as a map or dictionary with user-specified key type and data type. The declaration syntax is:

```
data_type name [index_type]
```

The data type can be every type which is allowed for fixed-size arrays. `index_type` is the data type of the index, or the wildcard index type (symbolized with *). "If * is specified, then the array is indexed by any integral expression of arbitrary size. An index type restricts the indexing expressions to a particular type." [5].

The elements of an associative array are created in case an entry is written to an index the first time. The order of the entries depends on the index type and is maintained by the array itself. Additionally several built-in methods are provided:

- The `function int num()`, returns the number of entries in the array.
- The `function void delete([input index])`, deletes the entry with the specified index, or all entries if no index is specified.
- The `function int exists(input index)`, returns whether an entry with this index exists or not.
- The `function int first(ref index)`, assigns the value of the first (smallest) index to the given variable index. The method returns 1 if the element exist, 0 otherwise.
- The `function int last(ref index)`, assigns the value of the last (largest) index to the given variable index. The method returns 1 if the element exist, 0 otherwise.
- The `function int next(ref index)`, assigns the value of the index next (greater) to the given index to the variable index. The method returns 1 if the element exist, 0 otherwise.
- The `function int prev(ref index)`, assigns the value of the index previous (smaller) to the given index to the variable index. The method returns 1 if the element exist, 0 otherwise.

## 4.3 Processes

SystemVerilog introduces three new specialized blocks for modelling combinational, latched or sequential logic hardware models. The `always_comb` block is used to model combinational logic. The sensitivity list is generated automatically from the contained statements. If a latched logic behaviour is intended, the `always_latch` should

be used. SystemVerilog also adds the `always_ff` block to indicate a sequential logic behavior. When using this specialized blocks the design intent is indicated to simulation, synthesis, and formal verification tools. Listing 4.3 gives a short example for each process. Also notice the `iff` in line 5, which allows to add condititions to the @ event control.

```
1 always_comb
2   a = b & c;
3 always_latch
4   if(ck) q <= d;
5 always_ff @(posedge clock iff reset == 0 or posedge reset)
6   acc <= reset ? 0 : reg + 1;
```

Listing 4.3: Specialized processes examples [?]

SystemVerilog also improves the multithreading options of Verilog. Figure 4.1 gives an overview of the features. There are three different ways of waiting for completion

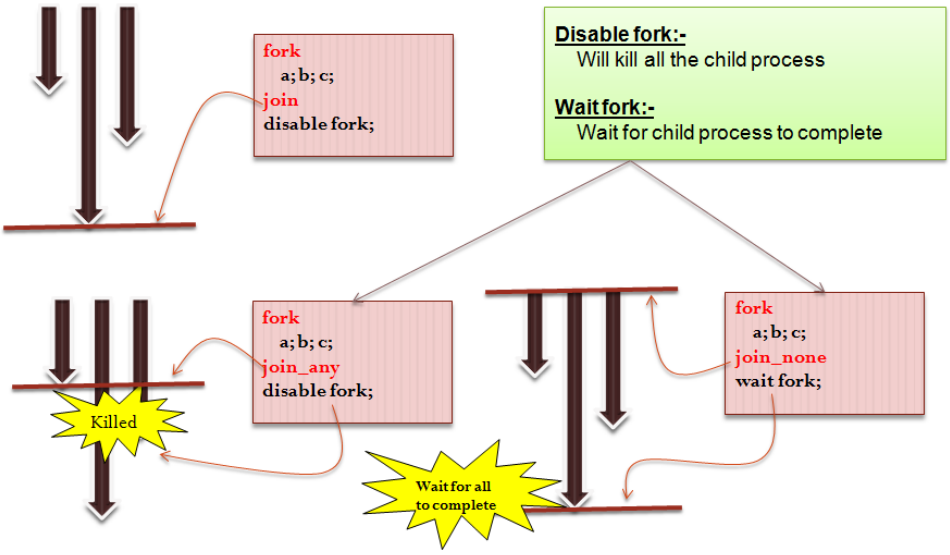


Figure 4.1: Overview over the multithreading features of SystemVerilog [2]

of the threads created by the fork:

- `join`, waits until all threads completed
- `join_any`, waits until at least one thread is completed
- `join_none`, doesn't wait on completion of any thread

In addition to these three join statements there are two statements which allow one thread to kill other threads or to wait on their completion. Where the `disable fork`

operation kills all running threads and the `wait fork` construct waits for the completion. If having named begin end blocks it is also possible to kill just one thread by calling `disable name_of_thread`.

## 4.4 Interface

The port model which is used in Verilog to connect modules is not productive for large modules with hundreds of ports. Therefore SystemVerilog introduces interfaces to allow the user to bundle ports. This also reduces the redundancy of port-name declarations. The keyword `modport` is used to restrict interface access within a module by indicating the directions of every variable declared inside the interface. Besides variables an interface also can have tasks and functions. They allow a more abstract level of modeling. A short example is given in Listing 4.4

```
interface intf;
  logic a;
  logic b;
  modport in (input a, output b);
  modport out (input b, output a);
endinterface

module top;
  intf i ();
  u_a m (.i1(i));
  u_b n (.i2(i));
endmodule

module u_a (intf.in i1);
endmodule

module u_b (intf.out i2);
endmodule
```

Listing 4.4: Interface example [17]

## 4.5 Classes

SystemVerilog offers an object-oriented programming model. This offers several new possibilities for the design of Test Benches. Like encapsulation, abstraction and a higher reusability. This chapter focuses on the features and models supported by the SystemVerilog class model. General knowledge about object-oriented programming can not be found in here, because this will go beyond the scope of this thesis.

Classes are declared with the keyword `class` and they can include attributes (variables) and subroutines (tasks and functions). SystemVerilog supports a single-inheritance model. Classes can be parameterized by type. This allows writing more generic classes which increases the reusability and avoids writing similar code for different sizes or types.

Polymorphism is done by labeling a method explicitly with the keyword `virtual`. By default all class attributes are declared public. To hide them they have to be marked either as `local` or `protected`.

Like in Java, garbage collection is done automatically. This means, that objects do not have to be destroyed explicitly.

Listing 4.5 presents a view examples.

```
1  module main;
2    class classA;
3      //Encapsulation
4      //public variable, visible everywhere
5      int public_var;
6      //protected variable, also visible in inherited classes
7      protected int prot_var;
8      //local variable, only visible inside the class
9      local int local_var;
10
11     //constructor
12     function new(int param);
13       local_var = param + 10;
14     endfunction
15
16     //a function
17     function int get_local_var();
18       return local_var;
19     endfunction
20
21     function void print();
22       $display("classA");
23     endfunction
24
25     //a virtual function
26     virtual function void vprint();
27       $display("classA");
28     endfunction
29
30     virtual function int add();
31       return local_var + prot_var + public_var;
32     endfunction
33   endclass
34
35   //inherited class with a type parameter (default type int)
```

```
36   class classB#(type T = int) extends classA;
37     T param_var;
38
39     function new(int param);
40       super.new(param);  //call constructor of base clase
41     endfunction
42
43     //override print function
44     function void print();
45       $display("classB");
46     endfunction
47
48     //override vprint function
49     virtual function void vprint();
50       $display("classB");
51     endfunction
52
53     //override add function
54     function T add();
55       T result;
56       //call add function of the baseclass
57       //and cast result to type T
58       $cast(result, super.add());
59       return result + param_var;
60     endfunction
61   endclass
62
63   initial begin
64     //object of type classA and param = 10
65     classA a = new(10);
66     //object of type classB with set T to real
67     classB #(real) b = new(20);
68
69     //create new object with param = 20
70     a = new(b.get_local_var());
71
72     //Polymorphism
73     a = b;
74     a.print();  //prints classA
75     a.vprint(); //prints classB
76     b.print();  //prints classB
77   end
78 endmodule
```

Listing 4.5: Object-oriented examples

## 4.6 Constrained random generation

Constrained random testing offers many advantages in comparison to a traditional, direct testing approach. At least with SystemVerilog both approaches can be combined easily.

Generally the course of actions looks like this. Variables can be declared random by use of the keywords `rand` or `randc`. Constraints restrict the legal values that can be assigned to the random variables. A solver called by the `randomize()` class method solves the constraints and generates legal random values. A simple example can be seen in Listing 4.6. If the structure of the Test Bench and the seed doesn't change, the generated random values will be the same in every run.

```
module main;
  class A;
    rand integer x;
    randc integer y;
    constraint c_x {x < 100;}
    constraint c_y {y > 10 && y < 90;}
  endclass

  initial begin
    A aobj = new();
    aobj.randomize();
    $display("x = %d; y = %d", aobj.x, aobj.y);
  end
endmodule
```

Listing 4.6: Simple constrained random generation example

The `randc` keyword defines a variable to be cyclic-random. This means that no value is repeated until every possible value has been assigned. Only bit and enumeration types can be declared to be `randc`. To reduce memory requirements, the maximum size of such variables should be limited.

### 4.6.1 Constraint blocks

Constraint blocks are used to restrict the possible values of a random variable. They are class members and must have a unique name inside the class. However, like external tasks and function bodies, constraint block bodies can also be declared outside a class. Because constraint blocks are class members they can be overridden by an inherited class, just like virtual tasks and functions. SystemVerilog offers various instruments which can be used to define constraints.

**Set membership** : The keyword `inside` is used to define a set of valid values. If the negation operator is set before the keyword, all values outside the set are valid. All values are uniformly distributed. Examples:

```
rand integer x,y;

//x has to be 0, 2, 10 to 20 or y to 2*y
constraint c {x inside {0,2,[10:20],[y:2*y]};}

//x has to be outside the given range
constraint c {!(x inside {1,3,10});}
```

Listing 4.7: Set membership

**Distribution** : Sometimes another then the uniform distribution is required. In this case the `dist` keyword can be used, to weight every value in the set differently. The solver sums the weights up and $value_i$ will get the probability $weight_i / weight_{sum}$. To assign a weight to a value there are the `:=` and the `:/` distribution operator. The difference is, if a weight is assigned to a range of values. The `:=` operator assigns the weight to every value in the range, whereas the `:/` assigns the weight to the whole range.

```
//the weights are 0 = 1; 2 = 2; 10, 11, 12 = 3; 20, 21, 22 = 1
//the probability for 0 = 1/15
constraint c {x dist {0 := 1, 2 := 2, [10:12] := 3, [20:22] :/ 3};}
```

Listing 4.8: Distribution

**If-else** constructs can be used inside a constraint to make the constraining of one variable depended on another. Example:

```
//If x is smaller 100, y has to be greater 100.
//Else, y has to be smaller 100.
constraint c {
  if(x < 100) {
    y > 100;
  } else {
    y < 100;
  }
```

Listing 4.9: If-else inside constraints

**Implications** are basically the same as using an if-else construct. However it has a shorter notation. Take care that if the condition is false, the generated random values will be unconstrained, see example below.

```
//If x is smaller 100, y has to be greater 100.
//If x is greater equal 100, y is unconstrained
constraint c {(x < 100) -> (y > 100);}
```

Listing 4.10: Implications

**Foreach loops** are used to constrain array elements. They can be used for every type of array (fixed-size, dynamic, associative, or queue). Listing 4.11 presents some examples.

```
rand byte A[];

//Every element of A has to be either 2,4,8 or 16
constraint c1 { foreach (A [i]) A[i] inside {2,4,8,16}; }

//Every element of A has to be greater than twice its index
constraint c2 { foreach (A [j]) A[j] > 2 * j; }

//Each array value is constrained to be greater than the preceding one
    .
//The implication prevents an out-of-bounds access.
constraint c3 { foreach (A [k]) (k < A.size - 1) -> A[k+1] > A[k]; }
```

Listing 4.11: Foreach loops, [5]

**Variable ordering:** The solver assigns the random values in a way that, "all combinations of legal values have the same probability of being the solution" [5]. However, this is not desirable in all cases. For example if a 1-bit variable should constrains a 32-bit data value, as it can be seen in Listing 4.13.

```
rand bit s;
rand bit [31:0] d;
constraint c { s -> d == 0; }
```

Listing 4.12: Variable ordering, [5]

Anyway, this will not lead to a 50% probability that $d == 0$. This is because `s` and `d` are determined together and that's why there are $2^{33}$ possible combinations. That is that the probability of $s == 1$ is only $1/2^{33}$. After adding:

```
constraint order { solve s before d; }
```

$s$ is solved separately and there $s == 1$ has a probability of 50%, sequentially 50% of the values of $d$ will be 0.

**Functions inside constraint blocks:** As it can be seen in Listing 4.14, it is also possible to use functions inside the constraint block.

```
rand int length;
rand bit [9:0] v;

//function which counts the 1's inside an array
function int count_ones (bit [9:0] w);
  for(count_ones = 00; w != 0; w = w >> 1)
    count_ones += w & 'b1;
endfunction

//length is constrained to the output of the function count_ones
constraint c1 {length == count_ones(v);}
```

Listing 4.14: Functions inside constraint blocks,[5]

### 4.6.2 Randomization Methods

The built-in method `randomize()` is used to generate random values for all random variables in the object, subjected to the constraints. If the randomization was executed successfully the method returns 1, otherwise 0.

By using `randomize() with constraint_block` constraints can be declared in-line. This offers greater flexibility because additional constraints can easily be added to a part of objects of the same class.

However, it is also possible to control whether a random variable is active or inactive. If a variable is set inactive it is not randomized by the solver. Disabling or enabling random variables is done with the `rand_mode()` method. Every random declared variable automatically has such a method. The syntax is: `object.variable.rand_mode(bit on_off)`.

In a similar way also constraints can be set active or inactive. Inactive constraints are not considered by the solver. The syntax is: `object.constraint_id.constraint_mode(bit on_off)`.

## 4.7 Coverage

Functional coverage answers the "Are we done?" question.

The specification of a coverage model is encapsulated in a coverage group. A covergroup includes one or more coverage points, cross coverage between coverpoints, optional formal arguments and coverage options. A sample is taken either on a specific event (e.g. `@(posedge clk)`) or manually by calling the `sample()` method of a covergroup. However, care is required to take samples only when data is meaningful. Coverage groups are defined between the keywords `covergroup` and `endgroup`. Instances of a covergroup are created with the `new` operator, just like instances of classes. It is possible to define covergroups in modules, programms, interfaces, or classes.

A coverage point contains a set of bins associated with its sampled values or its value-transitions. The bins can either be explicitly defined or created automatically by the system. A coverage point is defined with the keyword `coverpoint` and can have a optional label. Listing 5.2 demonstrates some examples.

```
typedef enum {IDLE, SEND, RECEIVE} state_t;
class A;
  logic [0:2] x;
  logic [0:9] y;
  state_t state;

  covergroup cov;
    //basic definition; the coverpoint has 8 bins, one for each
    //possible value of x; the coverpoint is labeled with "cp_x"
    cp_x: coverpoint x;

    coverpoint y {
      //creates 2 bins, one for the values 0 to 10 and one for 12
      bins b_a = { [0:10], 12 };

      //creates 20 bins one for each value;
      //b_b[100],...,b_b[109],b_b[300],...,b_b[309]
      bins b_b[] = { [100:109],[300:309]} };

      //explicitly creates 5 bins which covers the values
      //from 500 to 1000
      bins c_b[5] = {[500:1000]};

      //covers all other possible values of y
      bins d_b = default;
    }

    coverpoint state {
      //creates a bin for the transition IDLE to SEND
```

```
30        //and one for IDLE to RECEIVE
31        bins b_a[] = (IDLE => SEND, RECEIVE);
32      }
33    endgroup
34  endclass
```

Listing 4.15: Functional coverage examples

# 5 Universal Verification Methodology (UVM)

The Universal Verification Methodology (UVM) is a methodology to verify integrated circuit designs, standardised by Accellera. UVM is mainly based on the Open Verification Methodology (OVM), but also on VMM (Verification Methodological Manual) and eRM (e Reuse Methodology).

## 5.1 Concept

Basically UVM is a class library which provides base classes to build a reusable, constraint-random and coverage-driven Test Bench. The class library is implemented in SystemVerilog. "The UVM Test Bench architecture is modular to facilitate the reuse of groups of verification components either in different projects (horizontal reuse) or at a higher level of integration in the same project (vertical reuse)" [10]. Figure 5.1 shows the basic blocks of a UVM Test Bench. The communication between the blocks is done on transaction-level. This is done with Transaction-Level Modeling (TLM). Every component has a well defined TLM interface to communicate with other components over TLM channels. These encapsulates every component and isolates it from changes in other components. Thereby it is simple to replace one component by another, if the interface stays the same. Anyway, this leads to highly reusable verification components and Test Benches. The transition from TLM communication to pin level is done by drivers and monitors.

## 5.2 UVM Phases

In UVM, during execution of a test, every component runs through several phases. This process is controlled by the UVM phase controller. As soon as all components have finished a phase, the next one will be executed. Each component has a virtual

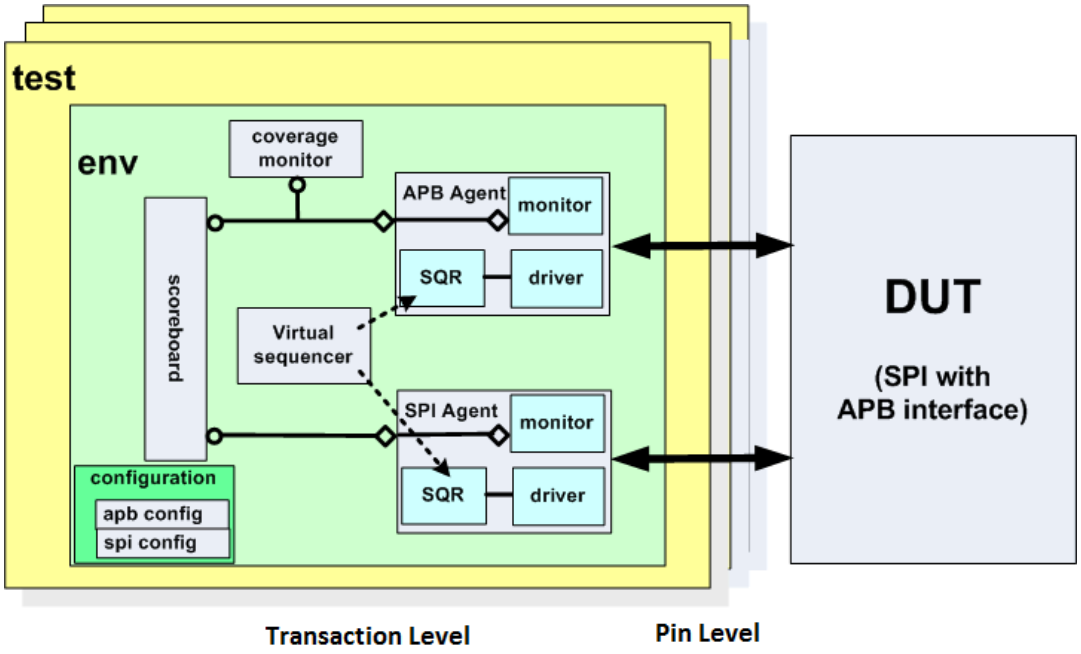


Figure 5.1: UVM Test Bench [10]

task or function for every phase. By overriding these callback functions, the developer fills them with appropriate functionality. It is not necessary to implement all phases in all components.

**build**
The construction of the Test Bench component hierarchy is done in this phase. Basically, at first the component is configured and afterwards all direct sub-components are created. This is the only phase which is, obviously executed top-down.

**connect**
In this phase the correspondent components are connected.

**end_of_elaboration**
At this point it can be assumed that the component hierarchy is created and all components are connected. This phase can be used to do some final adjustments.

**start_of_simulation**
The start_of_simulation phase is executed just before time zero. It can be utilized to

print some information, like the components hierarchy, or configuration information

**run**
This is the only simulation time consuming phase, therefore it is implemented as a task. Stimulus generation and checking of the activities of the Test Bench is done here. In parallel to this phase a bunch of phases is executed. Therefore, instead of the run phase, some of these can be implemented to reach a finer granularity.

**extract**
The extract phase is intended to retrieve and process information which was collected during run phase.

**check**
This phase is used to check the correctness of the extracted data.

**report**
In this phase the final reports are created and printed out or written to a report file.

**final**
Not yet finished operations are completed in this phase.

## 5.3 Components

This section describes the basic components and other interesting facilities offered by the UVM class library.

### 5.3.1 Factory

"The purpose of the UVM factory is to allow an object of one type to be substituted with an object of a derived type without having to change the structure of the Test Bench or edit the Test Bench code" [10]. To use the factory some coding conventions have to be fulfilled:

1. Each component or object must be registered at the factory. This can be done with one of the following registration macros, `‘uvm_component_utils(my_component)` for components or `‘uvm_object_utils(my_object)` to register objects.

2. To allow object creation inside the factory, the constructor of a component or object has to follow a prototype template and it should contain defaults for the constructor arguments. The constructor of `uvm_component` class is `function new(string name, uvm_component parent)` for `uvm_object` it is `function new(string name)`.

3. Instead of creating components or objects via the `new()` method, this is done with `m_comp = my_comp::type_id::create("my_component", this)` or `m_obj = my_obj::type_id::create("my_object")`.

To substitute an object with a derived one the static function `base_class::type_id::set_type_override(derived_class::get_type())` is used. The next time `base_class::type_id::create(...)` is called, an instance of the derived_class is returned.

### 5.3.2 Configuration database

The `uvm_config_db` class is the central place to save configurations or other information that are shared between several components or objects. Generally a top component (e.g. test) puts configurations into the database and later transactor components (e.g. driver) reads them out. A common use case is to pass virtual interfaces to the drivers and monitors.

To put something to the database the method

```
void uvm_config_db #(type T = int)::set(uvm_component cntxt,
                string inst_name, string field_name , T value);
```

is used. Where

- **T** is the type of the element being saved.
- **cntxt** and **inst_name** defines the scope in which the element is saved.
- **field_name** is the key of the new entry.
- **value** is the value of the new entry.

Similar to get an object from the database the method

```
bit uvm_config_db #(type T = int)::get(uvm_component cntxt,
          string inst_name, string field_name , ref T value);
```

is used. Where

- **T** is the type of the element being saved.
- **cntxt** and **inst_name** defines the scope in which the searched element is saved.
- **field_name** is the key of the searched entry.
- **value** will be the searched entry, if the get operation was successful.
- the return value is 1 if successful, 0 if not.

### 5.3.3 Driver

The driver is directly connected to the DUT and translates from TLM level to pin level. It receives commands (sequence items) from the sequencer, translates them and drives the appropriate signals. See Listing 5.3 for an example. A driver is derived from the library class `uvm_driver` and parameterised with the type of the sequence item it should handle. In the build phase the driver tries to get the virtual interface from the configuration database. However, the core functionality is implemented in the forever loop in the `run_phase` task. More precisely, in the example, the functionality is in the `send_cmd()` and also in the `send_clk()` method. First the driver tries to get a new sequence item from the sequencer. This method blocks until a new item is available. Next the item is translated and afterwards it is driven to the DUT. After that the driver signalises the sequencer that the execution of the current item is done. Optionally a response can be send back to the calling sequence.

```
class spi_driver extends uvm_driver#(spi_cmd);
  `uvm_component_utils(spi_driver);

  spi_agent_config m_cfg; // spi config
  virtual interface spi_if vif; // interface to spi

  //Constructor
  function new (string name="spi_driver", uvm_component parent=null);
    super.new(name, parent);
  endfunction : new

  //callback function for the build phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    //get configuration object from config db
```

```
16      if(!uvm_config_db #(spi_agent_config)::get(this,"","
            spi_agent_config",m_cfg))
17        `uvm_fatal("CONFIG_LOAD","Cannot get() ...");
18      vif = m_cfg.vif;
19    endfunction: build_phase
20
21    //callback function for the run phase
22    //start send_cmd and send_clk threads
23    virtual task run_phase(uvm_phase phase);
24      fork
25        send_cmd();
26        send_clk();
27      join
28    endtask : run_phase
29
30    //send_cmd
31    //get cmd from sequencer and send it to the dut
32    virtual protected task send_cmd();
33      ...
34      forever begin
35        //get new sequence item from the sequencer
36        //note: req is declared in uvm_driver
37        seq_item_port.get_next_item(req);
38        //drive item to the dut
39        vif.csn = 0;
40        ...
41        vif.csn = 1;
42        //signalise that the execution is finished
43        seq_item_port.item_done();
44        //optionally send a response back to the sequence
45        seq_item_port.put_response(rsp);
46      end
47    endtask : send_cmd
48
49    //create and drive the clock
50    virtual protected task send_clk();
51      real delay = (1000/(2*m_cfg.m_freq));
52      ...
53    endtask : send_clk
54  endclass : spi_driver
```

Listing 5.3: Driver example

### 5.3.4 Monitor

Like a driver, the monitor is also connected to the DUT. However, it is a passive component. The monitor observes the signal lines and converts the observations into

TLM level sequence items. These monitored items are provided to other components over the monitors analysis port. Listing 5.4 demonstrates a basic example. A monitor is derived from `uvm_monitor`. Basically the monitor runs in an endless loop, waiting on a new operation to start. Afterwards it creates a new sequence item and fills it with data, by collecting it from the signal lines. If the operation is finished, the item is written to the analysis port. This will call the callback method of all listeners on this port.

```
class spi_monitor extends uvm_monitor;
  `uvm_component_utils(spi_monitor);

  virtual interface spi_if vif; //interface to spi bus
  spi_agent_config m_cfg; //spi config
  uvm_analysis_port #(spi_cmd) ap; //analysis port
  spi_cmd m_item; //spi_cmd collected from the bus

  //Constructor
  function new (string name="spi_monitor", uvm_component parent=null);
    super.new(name, parent);
  endfunction : new

  //build phase
  function void build_phase(uvm_phase phase);
    ap = new("ap",this);
    if(!uvm_config_db #(spi_agent_config)::get(this,"","
        spi_agent_config",m_cfg))
      `uvm_fatal("CONFIG_LOAD","Cannot get() ...");
    vif=m_cfg.vif;
  endfunction: build_phase

  //run phase
  //collect spi commands and responses and create a spi_cmd out of
     that
  virtual task run_phase(uvm_phase phase);
    forever begin
      //get command
      m_item = spi_cmd::type_id::create({get_full_name(),".spi_cmd"});
      @(negedge vif.sclk);
      ...
      ap.write(m_item); //write item to the analysis port
    end
  endtask : run_phase
endclass : spi_monitor
```

Listing 5.4: Monitor example

### 5.3.5 Sequencer

A sequencers routes sequence items to a driver. They are generated by sequences. A sequencer can be supplied by more than one sequence. To do so, it holds a priority list of all sequence items which should be executed. A virtual sequencer is not directly connected to a driver. Instead it delegates sequence items or also sequences to a "real" sequencer. A sequencer is derived from the parameterised class `uvm_sequencer`. Generally all functionality of a sequencer is already implemented in this library class. Therefore usually a custom sequencer will look just like in Listing 5.5. The main purpose why writing a new class is to parameterise the base class with the type of the sequence item.

```
1 class spi_sequencer extends uvm_sequencer#(spi_cmd);
2     `uvm_sequencer_utils(spi_sequencer);
3     //constructor
4     function new (string name="spi_sequencer", uvm_component parent=
          null);
5       super.new(name, parent);
6     endfunction : new
7 endclass : spi_sequencer
```

Listing 5.5: Sequencer example

### 5.3.6 Agent

An agent encapsulates all modules needed to interact with a specific interface of the DUT. For example an SPI agent holds a sequencer which provides SPI items to a driver which is capable to drive these to the SPI interface of the DUT. The appropriate monitor observes the bin-level signals and creates TLM level SPI sequence items out of them. Additionally an agent can have an analysis component to perform checks. As well there is a configuration object which allows other components to configure the agent. As it can be seen in Listing 5.6 the base class is `uvm_agent`. At the build phase all components are created. They are connected to each other in the connect phase.

```
1 class spi_agent extends uvm_agent;
2   `uvm_component_utils(spi_agent);
3
4   spi_agent_config m_cfg;
5   // Components of the agent
6   uvm_analysis_port #(spi_cmd) ap; //Analysis port of the Agent
7   spi_driver m_driver; //spi driver
8   spi_sequencer m_sequencer; //spi sequencer
9   spi_monitor m_monitor; //spi monitor
10
```

```
11   //Constructor
12   function new (string name="spi_agent", uvm_component parent=null);
13     super.new(name, parent);
14   endfunction : new
15
16   //callback function for the build phase
17   function void build_phase(uvm_phase phase);
18     super.build_phase(phase);
19     if(!uvm_config_db #(spi_agent_config)::get(this,"","
         spi_agent_config",m_cfg))
20       `uvm_fatal("CONFIG_LOAD","Cannot get() ...");
21     // create the components of the agent
22     m_monitor = spi_monitor::type_id::create("spi_monitor", this);
23     //a passive agent only holds a monitor
24     if(m_cfg.active == UVM_ACTIVE) begin
25       m_sequencer = spi_sequencer::type_id::create("spi_seqr", this);
26       m_driver = spi_driver::type_id::create("spi_driver", this);
27     end
28   endfunction : build_phase
29
30   //callback function for the connect phase
31   function void connect_phase(uvm_phase phase);
32     //make the monitor analysis port visible outside of the agent
33     ap = m_monitor.ap;
34     if(m_cfg.active == UVM_ACTIVE) begin
35       //connect driver and sequencer
36       m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
37     end
38   endfunction : connect_phase
39
40 endclass : spi_agent
```

Listing 5.6: Agent example

### 5.3.7 Scoreboard

An essential part of a self-checking environment is a scoreboard. It verifies the operations collected by the monitors. The composition of a scoreboard depends on the DUT, but basically it compares actual and expected values and may record statistical information. The library base class is `uvm_scoreboard`.

### 5.3.8 Coverage monitor

A coverage monitor is used to collect functional coverage. This is done with coverage functionality supplied by SystemVerilog. It does not essentially have to be in a

separate class. The coverage can also be taken in the scoreboard or in the monitor or somewhere else.

### 5.3.9 Environment

The environment is somehow the top level class of the Test Bench. It holds agents for all interfaces of the DUT, a scoreboard and optionally a virtual sequencer and other analysis components like a coverage monitor. Furthermore there is a configuration object which allows tests to configure the environment. There is also a base class for environments, `uvm_env`. In the example in Listing 5.7 the environment holds two agents, a virtual sequencer and a scoreboard.

```
class cts_env extends uvm_env;
  `uvm_component_utils(cts_env);
  // Components of the environment
  epc_agent m_epc_agent;
  spi_agent m_spi_agent;
  cts_scoreboard m_sb;
  cts_virtual_sequencer m_vsqr;

  // Constructor
  function new(string name="cts_env", uvm_component parent=null);
    super.new(name, parent);
  endfunction : new

  //build phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    //create sub components
    m_epc_agent = epc_agent::type_id::create("epc_agent", this);
    m_spi_agent = spi_agent::type_id::create("spi_agent", this);
    m_sb = cts_scoreboard::type_id::create("scoreboard",this);
    m_vsqr = cts_virtual_sequencer::type_id::create("vsqr",this);
  endfunction : build_phase

  //connect phase
  function void connect_phase(uvm_phase phase);
    //the virtual sequencer holds instances of the sequencer
    //in the two agents.
    m_vsqr.m_epc = m_epc_agent.m_sequencer;
    m_vsqr.m_spi = m_spi_agent.m_sequencer;
    //connect the scoreboard to the analysis ports of the agents
    m_epc_agent.ap.connect(m_sb.epc_ap_imp);
    m_spi_agent.ap.connect(m_sb.spi_ap_imp);
  endfunction: connect_phase
endclass : cts_env
```

Listing 5.7: Environment example

### 5.3.10 Sequences

Sequences are used for stimulus generation. Figure 5.2 demonstrates the architecture of the sequence based stimulus generation. Generally a sequence creates sequence items, sends them via a sequencer to a driver. The driver converts the item into binary signals and drives them to the DUT. Optionally the driver sends a response back to the sequence.

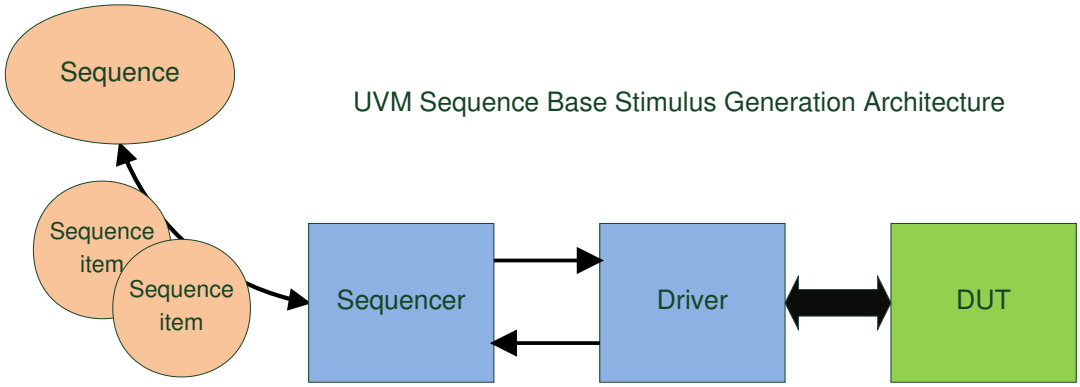


Figure 5.2: UVM Sequence Base Stimulus Generation Architecture [10]

Sequences are `uvm_objects`, therefore they have a shorter lifetime then components. Every sequence has a task called body. After the creation of a sequence, this task is executed and afterwards the sequence can be discarded. The body task is used to create and execute other sequences, or to create sequence items and send them to a driver, via a sequencer. The UVM base class is the parameterized class `uvm_sequence #(type REQ=uvm_sequence_item, type RSP=REQ)`. To execute a sub-sequence or send a sequence item, the `` `uvm_do() `` macros can be used. They support sequences and sequence items. In Listing 5.8 line 12 an item is sent to the driver and additional constraints for the randomization of the item are added. In line 19 a sub-sequence is executed. A virtual sequence only executes sub-sequences and does not send sequence items to a driver.

```
1  class spi_seq extends spi_base_seq;
2    `uvm_object_utils(spi_seq)
3    function new(string name = "spi_seq");
4      super.new(name);
5    endfunction : new
6    virtual task body();
7      spi_cmd cmd;
8      some_seq seq;
9      //use uvm macros to create a spi cmd, add some
10     //constraints, randomize it and send it via a
```

```
11      //sequencer to the driver
12      `uvm_do_with(cmd,
13      {m_cmd == SPI_WRITE;
14       m_opt_cmd == 'h0;
15       m_address == 'h0001;
16      })
17      //use a macro to execute a sub-sequence. This time after
18      //creation and randomization the body task is executed.
19      `uvm_do(seq)
20    endtask: body
21  endclass: spi_seq
```

Listing 5.8: Sequence example

### 5.3.11 Test

Each test class defines a test scenario for the Test Bench. A test creates the environment, configures the components by setting up configuration objects and starts sequences. It also configures which version of a component should be created by overriding the specific entry in the factory. Typically one will create a base test class in which, among others, the environment is created, configuration objects declared and maybe a base sequence is started. Derived test classes then configures the components and chooses which sequence should be started. Listing 5.9 shows parts of a base class as well as of a derived test class. In the `spi_base_test` class the environment is created and also the configuration object. Also the interfaces are fetched. In the `spi_test` class the attributes of the configuration object are set to a specific value. Also the base sequence is overwritten in the factory. Therefore the create method, called in in the `end_of_elaboration_phase()` callback method, will return an instance of `spi_seq`. The sequence then is executed during the `run_phase()`.

```
1  class spi_base_test extends uvm_test;
2    ...
3    cts_env m_env;
4    spi_base_seq m_seq;
5    spi_agent_config m_spi_agent_cfg;
6    // Build phase of the test
7    function void build_phase(uvm_phase phase);
8      super.build_phase(phase);
9      //create environment and configuration object
10     m_env = cts_env::type_id::create("env",this);
11     m_spi_agent_cfg = spi_agent_config::type_id::create("spi_at_cfg");
12     //get interface from config db
13     if(!uvm_config_db #(virtual spi_if)::get(this, "", "SPI_vif",
           m_spi_agent_cfg.vif))
14       `uvm_fatal("CONFIG_LOAD","Cannot get() ...");
15   endfunction : build_phase
```

```
16
17    virtual function void end_of_elaboration_phase(uvm_phase phase);
18      //create main sequence
19      m_seq = spi_base_seq::type_id::create("top_seq");
20      //randomize the sequence
21      m_seq.randomize();
22    endfunction : end_of_elaboration_phase
23
24    // Run phase of the test
25    virtual task run_phase(uvm_phase phase);
26      phase.raise_objection(this);
27      //start sequence on the sequencer
28      m_seq.start(m_env.m_spi_agent.m_sequencer);
29      phase.drop_objection(this);
30    endtask : run_phase
31  endclass : spi_base_test
32
33  class spi_test extends spi_base_test;
34    ...
35    virtual function void build_phase(uvm_phase phase);
36      super.build_phase(phase);
37      //configure spi agent
38      m_spi_agent_cfg.m_cpol = 1;
39      m_spi_agent_cfg.m_freq = m_spi_agent_cfg.std_freq / 2;
40      //write configuration objects into the db
41      uvm_config_db#(spi_agent_config)::set(this, "*", "spi_agent_config
          ", m_spi_agent_cfg);
42      //specify that a sequence of type spi_seq should be executed
43      spi_base_seq::type_id::set_type_override(spi_seq::get_type());
44    endfunction
45  endclass : spi_test
```

Listing 5.9: Test example

### 5.3.12 Top module

At least one static module is needed to connect the Test Bench with the DUT and to create a test object. To start a test the method `run_test()` is used. There are two ways to specify which test should be started. One possibility is to call the method with the class name of the test as an argument. A more flexible option is to use the plus-argument `+UVM_TESTNAME=test_name` at the simulator call. However which possibility is used, the UVM environment automatically creates an object of the specified test class via the factory.

# 6 Device Under Test (DUT)

The aim of the Comprehensive Transponder System (CTS) called DUT is to have an RFID platform which supports multiple frequencies and multiple standards. But still is a small passive tag. At the moment the systems supports the EPC standards for High Frequency (HF) and Ultra High Frequency (UHF) as well as the Near Field Communication (NFC) standards ISO/IEC 14443 and ISO 15693.

The CTS project was started some years ago and is currently getting a larger redesign. Nevertheless,

"The ambition of the project is the integration of multiple radio frequency carrier- and thus multiple standard technologies into one transponder system without increasing cost and without compromising performance. Although the comprehensive transponder system supports multiple standards and technologies it may consist only of one simple antenna and a very small chip both connected using just two pins like it is the case for existing transponders" [13].

Figure 6.1 shows a basic block diagram of the system. Besides the two contactless interfaces, there is a sensor interface to allow the connection of sensors, like a temperature or a Hall effect sensor. Additionally the system has an SPI slave interface which allows a simpler configuration. Also there is a NVM in form of an Electrically Erasable Programmable Read-Only Memory (EEPROM). Additionally the platform includes a SPI master interface which allows the communication with other devices. All these blocks are connected via a Wishbone Bus. The light grey colored blocks are used by the Test Bench.

The EPC module supports the standards EPC Class-1 Generation-2 UHF as well as EPC Class-1 HF as they are described in chapter 2. It is divided in two parts. The analog part is doing for instance the modulation and demodulation, but also detects if the signal is HF or UHF. The digital part implements the protocol with the help of a small RISC processor. An older version, implemented as a classic state machine, can be seen in [8] and [13].

The SPI slave interface can be used to configure the device and offers a direct and faster access to the NVM. It is basically described in chapter 3. The SPI master uses another data format and protocol. It is not described in this thesis because it is not used.

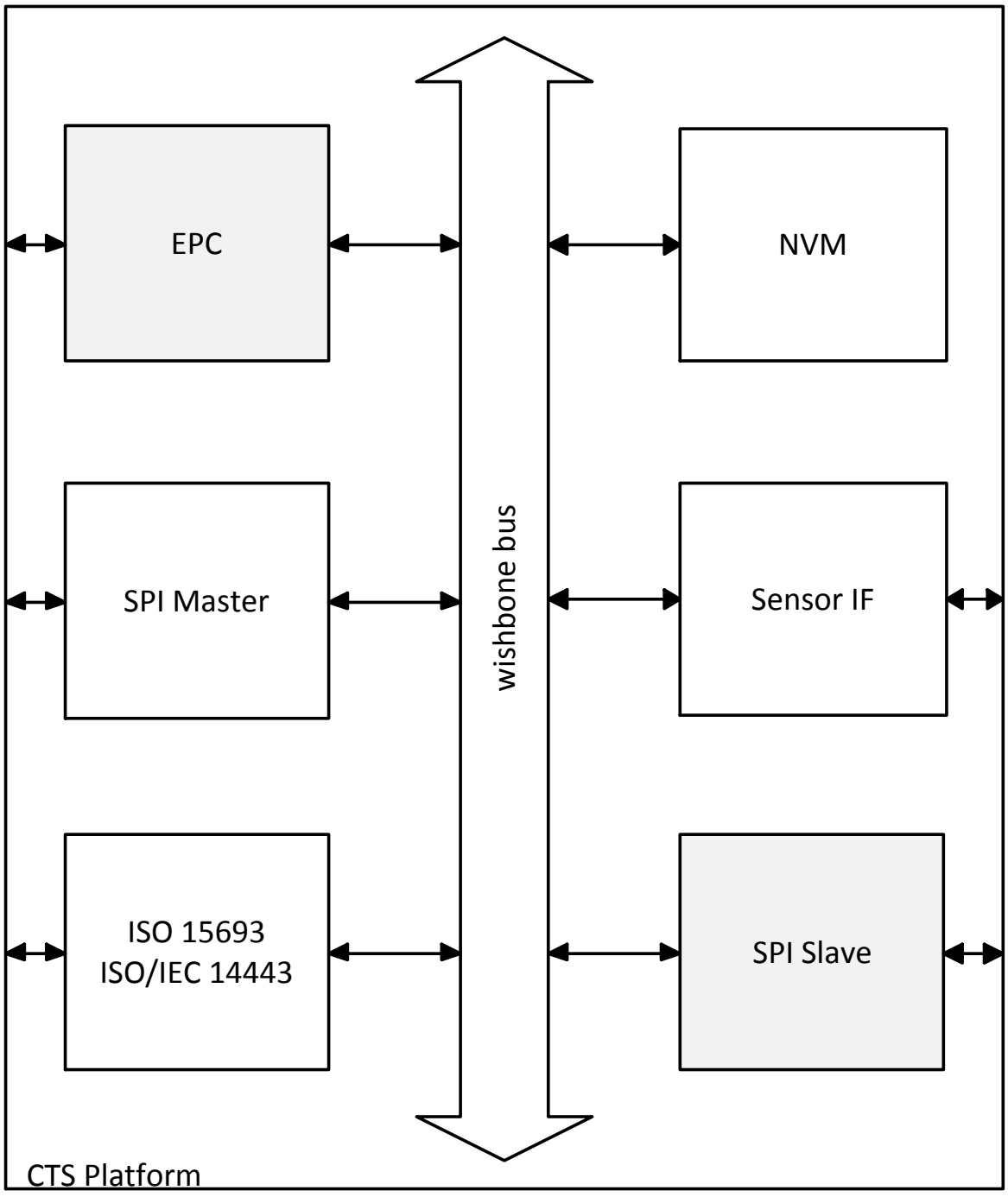


Figure 6.1: CTS Platform Block Diagram

# 7 Test Bench

In this chapter the Test Bench is discussed. It is written in SystemVerilog with using the UVM class library to allow a constraint-random testing. The primary goal was to write a Test Bench for the EPC block which is getting a large redesign. Later on an agent for the, also redesigned, SPI slave module was added.

## 7.1 Overview

Figure 7.1 gives a general overview over the classes and modules of the Test Bench and the connected DUT.

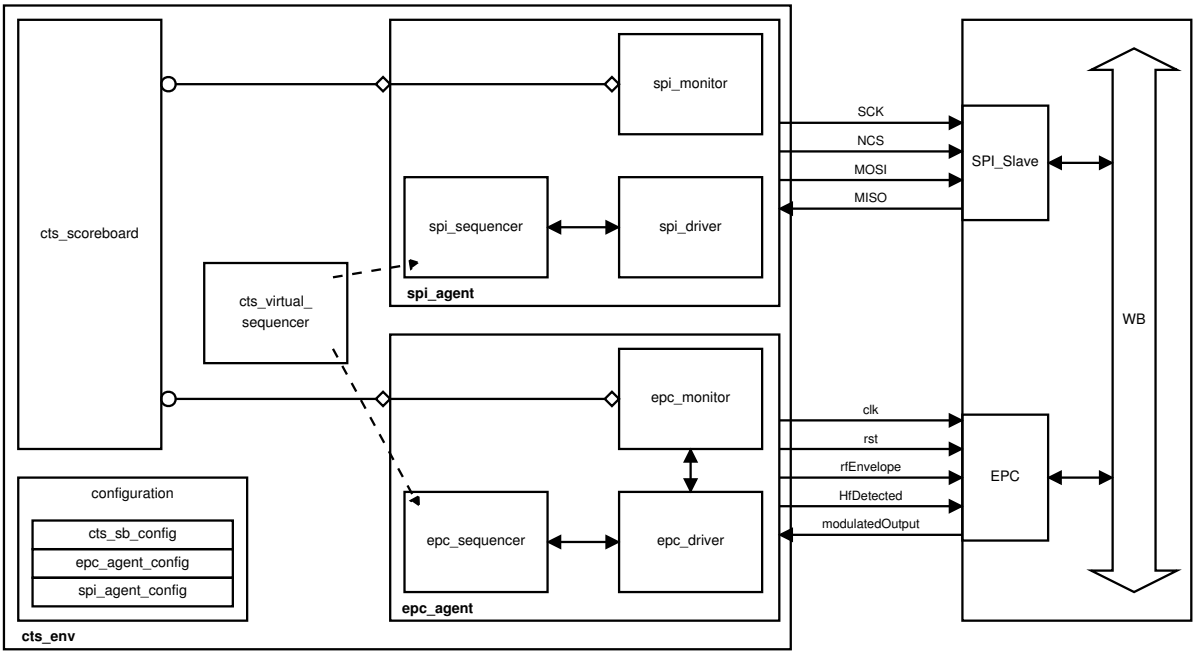


Figure 7.1: Overview blockdiagram of the Test Bench

On the left side is the Test Bench. The top level class is called `cts_env`. It contains an agent which handles with the SPI interface (`spi_agent`), as well as one for the

EPC interface (epc_agent). They are connected with the appropriate module of the DUT on the right hand side. This is done via SystemVerilog interfaces (they are not printed as blocks in the figure). The EPC agent is connected directly to the digital part of the EPC module. The signals can be seen in the figure and are described in the following:

- clk: System clock
- rst: System reset
- rfEnvelope: Encoded data input stream
- HfDetected: HF or UHF mode
- modulatedOutput: Encoded data output stream

The SPI agent is connected to the slave via the signals described in chapter 3. The agents are configured with either the epc_agent_config or the spi_agent_config configuration object.

Furthermore the environment holds a virtual sequencer to allow sequences to make use of both agents. Last but not least the scoreboard holds a model of the DUT to perform checks and functional coverage. The scoreboard is configured with the cts_sb_config configuration object.

## 7.2 EPC Agent

The epc_agent class contains all the components that are needed to drive EPC commands to the EPC module and to collect the responses. The epc_driver catches commands (epc_cmd) from the epc_sequencer, converts them and drives the signal lines. The epc_monitor receives the symbols, converts them to sequence items (epc_cmds also contains the response) and writes the items to the analysis port. Additionally the monitor signalises the driver when the response is finished by putting the collected item onto a separate transaction channel. In some cases (e.g. QUERY commands) the driver returns the response back to the calling sequence over the response channel of the sequencer.

The agent is configured by setting the attributes of a epc_agent_config configuration object and saving it afterwards into the config db. The following attributes can be set:

- hf_uhf: Use HF or UHF mode
- freq: System frequency

- tari: reference time Tari
- times: A structure for the values of the length of data-0, data-1, delimiter, RT-Cal, TRCal and PW.

Additionally the object offers two methods to allow an easier and faster configuration. The `set_std_values(hf_uhf_e hf_uhf)` sets all other attributes to some standard values dependent on the mode. With `set_std_times(hf_uhf_e hf_uhf, real tari)` also the length of tari can be defined. The other times are set as multiples of tari.

### 7.2.1 Commands

The various EPC commands are implemented as sequence items. All commands are inherited from the base class `epc_cmd`. Figure 7.2 figures the inheritance diagram for `epc_cmd` and gives an overview of all implemented commands.

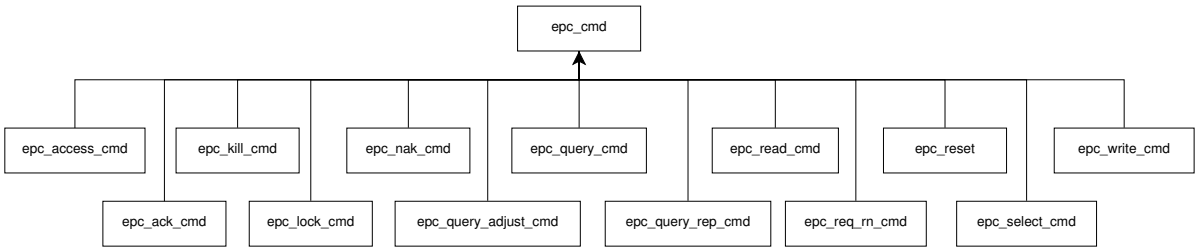


Figure 7.2: Inheritance diagram for epc_cmd

The base class has three basic variables and several virtual methods. The members are:

- `epc_cmds_e m_cmd`: An enumeration type which represents the command code in a readable form.
- `logic m_req_data[]`: The remaining part of the command in form of a bit stream (dynamically allocated).
- `logic m_rsp_data[]`: The possible response in form of a bit stream (dynamically allocated).

All the three variables are declared `protected`, so they can only be manipulated from inside of this or a inherited class.

The methods offers only rudimentary functionality and some of them will be usually overwritten in a specialised class.

- `set_req_data(logic data[])`: This method sets the fields of the command out of a bit stream. It necessarily has to be overwritten in a specialised command class. Typically the monitor uses this method after reading the symbols and converting them into a bit stream to fill the request part of a command object with contents. However, in the base method only `m_req_data` is set to `data`.
- `set_rsp_data(logic data[])`: This is the equivalent of the prior method for the response fields of a command object.
- `get_req_data(ref logic data[])`: The opposite of `set_req_data()` is done by this method. The driver then uses this method to generate a bit stream to send it to the DUT afterwards.
- `set_rsp_data(ref logic data[])`: This method simply returns `m_rsp_data` and is not declared as virtual.
- `is_valid_cmd()`: The scoreboard uses this method to perform a basic check if a valid command was transmitted. In the base class the method proves if the length of the command (without the command code) is larger than a minimum size of four bits.
- `is_valid_rsp()`: Similar to the previous method, this one checks if the response is valid. In `epc_cmd` this method does not have a functionality, it always returns false. Therefore classes of commands with responses must override this method.
- `get_cmd()`: Simple returns `m_cmd`;
- `get_cmd_name()`: Returns the name of the command as a string.

The Query command serves as an example of how a command is implemented. The other commands are basically realized alike. Listing 7.1 shows the `epc_query_cmd` class.

As it can be seen at the lines 8 to 14 all fields of the command are declared as to be random. There are no constraints inside the class. If desired they are defined at object creation. The response fields are declared at lines 17 and 18. Understandably they are not random.

At the lines 28 to 40 a realization of the `set_req_data()` method can be seen. The bit stream saved in `data` is first converted into a packed array, because that makes it easier to assign the appropriate parts to the fields.

The `set_rsp_data()` method does the same for the response fields. The checking at row 48 is needed if a FM0 encoding is used (see also section 7.2.3). Because the crc5 field is only present in HF mode the testing at line 51 is needed.

At the rows 58 to 68 the creation of a bit stream out of the command fields is demonstrated. The command code and all other fields are concatenated to one packed array. Because `m_req_data` is an unpacked array, the `foreach` loop is needed to convert `tmp` to an unpacked array. Afterwards a 5 bit long CRC is generated and appended to the bit stream.

The lines 71 to 83 presents the check methods `is_valid_cmd()` and `is_valid_rsp()`. They simply prove if the command or the response bit stream are of valid length and if the crc check is correct.

```
1  class epc_query_cmd extends epc_cmd;
2    `uvm_object_utils(epc_query_cmd);
3
4    localparam static_size=17;
5    localparam cmd_code='b1000;
6
7    //command fields
8    rand logic m_dr;
9    rand logic[0:1] m_m;
10   rand logic m_tr_ext;
11   rand logic[0:1] m_sel;
12   rand logic[0:1] m_session;
13   rand logic m_target;
14   rand logic[0:3] m_q;
15
16   //response fields
17   logic[0:15] m_rn16;
18   logic[0:4] m_crc5;  //only for HF
19
20   // Constructor
21   function new(string name = "epc_query_cmd");
22     super.new(name);
23     m_cmd = QUERY;
24   endfunction : new
25
26   // set_req_data: sets the fields of the command out of a bitstream
27   //  input data: bitstream
28   virtual function void set_req_data(logic data[]);
29     logic[0:static_size-1] tmp;
30
31     super.set_req_data(data);
32     for (int i=0; i<static_size; i++) tmp[i] = data[i];
33     m_dr = tmp[4];
34     m_m = tmp[5:6];
35     m_tr_ext = tmp[7];
36     m_sel = tmp[8:9];
37     m_session = tmp[10:11];
38     m_target = tmp[12];
39     m_q = tmp[13:16];
40   endfunction: set_req_data;
```

```
41
42    // set_rsp_data: sets the fields of the response out of a bitstream
43    //  input data: bitstream
44    virtual function void set_rsp_data(logic data[]);
45      super.set_rsp_data(data);
46
47      //handle dummy-1 if it is still in the data
48      if (m_rsp_data.size() == 17 || m_rsp_data.size() == 22)
49        m_rsp_data=new[m_rsp_data.size()-1](m_rsp_data);
50      for(int i=0; i<16; i++) m_rn16[i] = m_rsp_data[i];
51      if (m_rsp_data.size() > 17)
52        for(int i=16; i<21; i++) m_crc5[i-16] = m_rsp_data[i];
53    endfunction: set_rsp_data;
54
55    // get_req_data
56    // creates a bitstream out of the command fields and generates crc
57    //  output data: bitstream
58    virtual function void get_req_data(ref logic data[]);
59      logic[0:static_size-1] tmp;
60      logic crc[] = new[5];
61
62      m_req_data = new[static_size];
63      tmp={cmd_code,m_dr,m_m,m_tr_ext,m_sel,m_session,m_target,m_q};
64      foreach (m_req_data[i]) m_req_data[i] = tmp[i];
65      helpers::generate_crc(crc, m_req_data);
66      m_req_data = new[m_req_data.size()+crc.size()]({m_req_data,crc});
67      super.get_req_data(data);
68    endfunction: get_req_data;
69
70    // is_valid_cmd: simple check if the cmd is valid
71    virtual function bit is_valid_cmd();
72      return m_req_data.size()==22&&helpers::is_valid_crc(m_req_data,5);
73    endfunction: is_valid_cmd;
74
75    // is_valid_rsp: verifies rsp
76    virtual function bit is_valid_rsp();
77      if (m_rsp_data.size() == 16)
78        return 1;
79      if (m_rsp_data.size() == 21) begin
80        return helpers::is_valid_crc(m_rsp_data,5);
81      end
82      return 0;
83    endfunction: is_valid_rsp;
84  endclass : epc_query_cmd
```

Listing 7.1: Realization of the Query command

### 7.2.2 Driver

The driver has two main tasks to do. First creating the clock and sending it to the DUT. Second getting EPC commands from the sequencer, translating them and driving the signal lines.

First the driver initializes the DUT by setting the reset line low and then high again. Next the HfDetected_i line is set appropriate to the configuration. After a new sequence item is available the driver sends it to the DUT. How this is done can be seen in Listing 7.2.

```
virtual protected task send_cmd();
  logic data[];
  epc_query_cmd query;

  //send preamble/frame_sync
  if (req.get_cmd() == QUERY) begin
    send_preamble();
    //update the times
    $cast(query,req);
    if (m_cfg.m_hf_uhf == UHF)
      m_calc_times.calc_std_times_uhf(query.m_dr);
    else
      m_calc_times.calc_std_times_hf(query.m_dr);
  end
  else
    send_frame_sync();

  //send PIE encoded data
  req.get_req_data(data);
  for (int i=0; i<data.size(); i++) begin
    vif.rfEnvelope_i = 1;
    if (data[i] == 0)
      #(m_times.data_0 - m_times.pw);
    else if (data[i] == 1)
      #(m_times.data_1 - m_times.pw);
    vif.rfEnvelope_i = 0;
    #(m_times.pw);
  end
  vif.rfEnvelope_i = 1;

  //wait the appropriate time until sending the next cmd
  if (req.get_cmd() == SELECT) begin
    #m_calc_times.T4;
  end
  else begin
    received_rsp = 0;
    fork
      begin : wait_for_rsp
```

```
39          @(posedge vif.modulatedOutput_o);
40          received_rsp = 1;
41        end
42        begin : wait_for_T1
43          if (req.get_cmd() == WRITE || req.get_cmd() == LOCK || req.
                get_cmd() == KILL)
44            #20ms;
45          else
46            #m_calc_times.T1_max;
47        end
48      join_any
49      disable fork;
50
51      if (received_rsp) begin
52        rsp_finished = 0;
53        wait(rsp_finished);
54        #m_calc_times.T2_min;
55      end
56      else
57        #m_calc_times.T3;
58    end
59    #10;
60  endtask: send_cmd
```

Listing 7.2: Send command task

First, if the command is a QUERY, instead of a a frame_sync a preambel has to be sent. Also the link timing could have been changed, because the dr parameter is set in a QUERY. Therefore these times must be updated.

Next, the command is PIE encoded and sent to the DUT. This is simply done by setting rfEnvelope_i high for the appropriate time.

Afterwards the driver has to wait a proper time until sending the next command is allowed. If it was a SELECT it is simply the time T4. Else, either there is a response within a time interval or not. Depending on the command the interval is either T1 or 20 *ms*. In case of a received response, the driver must wait until it is finished (signalised by the monitor) and the time T2 afterwards. Otherwise the interval to wait is the time T3.

If it was one of the QUERY commands or a REQ_RN the driver puts the response on the response channel and afterwards it is ready to handle with a new command.

### 7.2.3 Monitor

The monitor reads commands and responds from the signal lines, decodes them and creates sequence items. The sequence items are then provided to the scoreboard.

A negative edge on rfEnvelope_i indicates the start of a new command. The preamble or the frame sync is skipped. The PIE encoded symbols are encoded by measuring the time between two positive edges. If the time interval is shorter than a pivot ($RTCall/2$), the symbol represents a '0' else it is a '1'. This is done until the time interval is larger then the duration of a data-1 symbol. After the bit stream is collected, the command code is decoded and the appropriate sequence item is created and filled with data. If the command is a QUERY, the parameters dr, tr_ext and M are saved, because they are needed for the correct decoding of the response. Also the link times (T1, T2, ...) are calculated.

Afterward the monitor waits until a new command or a response appears. If it is a response, the time between the last rising edge of rfEnvelope_i (end of command) and the first falling edge of modulatedOutput_o (begin of the response) is measured and verified. If the interval was to long, an error message is printed. Next the symbols are decoded. Depending on the M parameter and the mode (HF or UHF), the response is either FM0, Miller or Manchester encoded.

#### FM0 encoding

The signal is sampled with twice the BLF. The sampling continues until there are more then three consecutive bits with the same value. Three bits with the same value are only possible in the preamble. The monitor also proves if the symbol duration is inside the allowed range.

This sampled data stream is now used to decode the symbols. A '0' is represented by two different sampled values, a '1' by two equal ones. Listing 7.3 shows the source code of the decoding. In line 7 it is checked that there is a phase inversion between two consecutive symbols.

```
1  data = new[(data_en.size()-count)/2];
2  for(int i=0; count<data_en.size(); count+=2, i++) begin
3    if (data_en[count] == data_en[count+1])
4      data[i] = 1;
5    else if (data_en[count] != data_en[count+1])
6      data[i] = 0;
7    if (data_en[count-1] == data_en[count]) begin
8      data[i] = 1'bx;
9      `uvm_error("FM0 sequence error","No phase inversion occurred!")
10   end
11 end
```

Listing 7.3: FM0 decoding

Also the correctness of the preamble is proved, before. A Problem is that in some cases the dummy-1 is included at the end and in other cases not. The first one happens when the dummy-1 is represented by a '11', the second one by a '00'. Therefore this is handled later in the `epc_cmd`.

### Miller encoding

The signal is also sampled with twice the BLF, but it continues only until there are more then two successive equal values. The BLF is the frequency of the sub-carriers, this means that the symbol rate is either half, a quarter or an eighth of BLF.

The decoding can be seen in Listing 7.4. A '0' is represented by a 2*M long zero-one sequence (of course also a one-zero). A '1' is detected by having the same value at the beginning and the end of the sequence and consists of two zero-one sequences in the first and second half (see also lines 12 to 16).

```
1  data = new[(data_en.size()-count)/(2*M)];
2  for(int i=0; count<data_en.size(); count+=(2*M), i++) begin
3    if (is_a_one(data_en, count, 2*M))
4      data[i] = 1;
5    else if (is_one_zero(data_en, count, 2*M))
6      data[i] = 0;
7    else begin
8      data[i] = 1'bx;
9      `uvm_error("MILLER sequence error","Incorrect encoded!")
10   end
11 end
12 protected function bit is_a_one(logic data[], int from, int length);
13   return data[from] === data[from+length-1]
14            && is_one_zero(data, from, length/2)
15            && is_one_zero(data, from+length/2, length/2);
16 endfunction: is_a_one
```

Listing 7.4: Miller decoding

Certainly also the preamble is checked upon correctness. With Miller encoding, there is no problem to detect and remove the dummy-1.

### Manchester encoding

The signal is still sampled with twice the BLF. The process pursues until more then $2 * N_{sc} + 1$ (Number of sub-carriers) values are equal. This maximum number of the same values happens at a data-0, data-1 sequence.

The following decoding is done as it can be seen in Listing 7.5. If the first half of a symbol is zero and the second half a one-zero sequence, the symbol represents a '1'. If it is in the inverse order, it is a '0'.

```
data = new[(len-count)/(2*M)];
for(int i=0; count<len; count+=(2*M), i++) begin
  if (is_zero(data_en, count, M) && is_one_zero(data_en, count+M, M))
    data[i] = 1;
  else if (is_one_zero(data_en, count, M) && is_zero(data_en, count+M,
      M))
    data[i] = 0;
  else begin
    data[i] = 1'bx;
    `uvm_error("MANCHESTER sequence error","Incorrect modulated!")
  end
end
```

Listing 7.5: Manchester decoding

The preamble and the end-of-file are checked as well.

After decoding the data is saved in the sequence item (the EPC command) and published over the analysis port. Besides also the driver is informed that the transmission is finished now.

## 7.3 SPI Agent

All components needed to communicate with the SPI interface are combined in the `spi_agent` class. The `spi_sequencer` supplies sequence items (the `spi_cmds`) to the `spi_driver`. The driver translates the items into signals and drives them on the lines. The `spi_monitor` monitors the signal lines, creates an `spi_cmd` and offers it over a analysis port to the scoreboard.

The agent is configured over a `spi_agent_config` configuration object. The following attributes can be set:

- `m_freq`: Frequency of the SPI clock (default 366 kHz).
- `m_write_wait_time`: Time to wait after a write command (default 3 *ms*, because the NVM write is very slow).
- `m_read_wait_time`: Time to wait after a read command (default 15 $\mu s$).
- `m_cpol`: Value of the clock at idle (default '1').

However, if only the SPI interface is tested, the EPC agent is still needed. This is because the EPC driver generates the system clock and reset for the tag. The frequency for the SPI only case is 2.2 MHz.

### 7.3.1 Commands

Similar to EPC there is a base class, called `spi_cmd` and an inherited class for each of the three commands WRITE, READ and STATUS. The base class has the following attributes, which are needed by all commands.

- `m_cmd`: Determines the type of the command. It is of the enumeration data type `spi_cmds_e`, which has three different values, SPI_WRITE, SPI_READ and SPI_STATUS. The field also represents the instruction code. The command type is declared protected.
- `m_opt_cmd` is for the optional NVM command code

The methods offers basic functionality and most of them are usually overwritten or rather enhanced in the inherited classes to handle with the additional attributes.

- `get_cmd()`: returns m_cmd since it is protected
- `get_cmd_name()`: returns the command type in a readable form
- `get_req_data(ref logic data[])`: Creates the bit stream of the request out of the attributes of the command. In the base class the method creates the instruction word. The specialised classes appends the other words, if present.
- `set_req_data(logic data[])`: Does the opposite of the prior method.
- `set_rsp_data(logic data[])`: Sets the response attributes out of a bit stream.
- `get_rsp_length()`: Returns the length of the response. Primary this is needed by the driver.
- `do_status_compare(spi_cmd item)`: This method is used by the scoreboard to prove if the response to a STATUS command is valid. It only has a functionality implemented in `spi_read_cmd` and `spi_write_cmd`.

Besides the first two methods all other are declared virtual to allow polymorphism.

As an example for a inherited class the `spi_read_cmd` has two additional attributes, `m_address` to for the address to read from and `m_rdata` for the responded requested data.

In the `get_req_data()` method at first the method of the base class is called. Afterwards the address is added to the bit stream. `set_req_data()` works in the same principle. The command fields are set by the super method, afterwords the address field is set. In the `set_rsp_data()` method the responded data is assigned to the attributes. `get_rsp_length()` returns the length of the response in bits, which is 16 in this case. `do_status_compare` compares the response of a STATUS command with the attributes of this object and prints errors in case some are not equal.

### 7.3.2 Driver

The driver has to generate the SPI clock as well as to transmit the commands to the DUT.

The clock generating task supports enabling and disabling of the clock signal. As well as different idle levels (CPOL). This is defined in the configuration object.

The send command task is presented in Listing 7.6. The driver starts at the moment the system reset is deasserted. After a new request is available from the sequencer the transmission starts. First the SPI slave is selected and the clock is enabled. Next the sequence item is transformed into a bit stream (line 14). Afterwards the bits are driven onto the MOSI signal line. Depending on the CPOL parameter, this is done either on the rising or the falling edge of SCLK.

If it is a READ the driver now waits a configured time the tag needs to get the response ready. After that the response is clocked out. The information about the length of the response is supplied by the sequence item. In case of a WRITE the driver has to wait for a configured time until the tag is ready again. During waiting times the clock is disabled. At last the chip is deselected and the driver waits until a new command is available.

```
1  virtual protected task send_cmd();
2    logic req_data[];
3
4    #50ns;
5    @(posedge vif_epc.rst_i);
6    vif.mosi = 0;
7    vif.csn = 1;
8    forever begin
9      seq_item_port.get_next_item(req);
10     vif.csn = 0;
11     #400ns;
12     clk_on = 1;
13
14     req.get_req_data(req_data);
15     //send command
16     foreach(req_data[i]) begin
17       @(negedge vif.sclk iff m_cfg.m_cpol == 1
18         or posedge vif.sclk iff m_cfg.m_cpol == 0);
19       vif.mosi = req_data[i];
20     end
21
22     //wait until rsp is finished
23     @(posedge vif.sclk iff m_cfg.m_cpol == 1
24       or negedge vif.sclk iff m_cfg.m_cpol == 0);
25     if(req.get_cmd() == SPI_READ) begin
26       clk_on = 0;
```

```
27        #m_cfg.m_read_wait_time
28        clk_on = 1;
29      end
30      for(int i=0; i<req.get_rsp_length(); i++) begin
31        @(posedge vif.sclk iff m_cfg.m_cpol == 1
32          or negedge vif.sclk iff m_cfg.m_cpol == 0);
33      end
34      clk_on = 0;
35      if(req.get_cmd() == SPI_WRITE)
36        #m_cfg.m_write_wait_time;
37      else
38        #500ns;
39
40      vif.csn = 1;
41      seq_item_port.item_done();
42    end
43  endtask : send_cmd
```

Listing 7.6: SPI driver, send command task

### 7.3.3 Monitor

The monitor reads SPI commands and their possible response and combines them to a sequence item. The item is then provided to the scoreboard over an analysis port.

At first the monitor reads the instruction word to determine which type of SPI command should be created. Afterwards the appropriate object is created and the `req_data` and `rsp_data` arrays are resized to the required length. Next the remaining part of the request (if exist) is read from MOSI. After this the response is read from MISO. With the help of the set methods the sequence item is filled with data and than published over the analysis port.

```
1  virtual task run_phase(uvm_phase phase);
2    logic req_data[], rsp_data[];
3
4    forever begin
5      //get command
6      req_data = new[16];
7      @(negedge vif.sclk iff m_cfg.m_cpol == 1
8        or posedge vif.sclk iff m_cfg.m_cpol == 0);
9      foreach(req_data[i]) begin
10       @(posedge vif.sclk iff m_cfg.m_cpol == 1
11         or negedge vif.sclk iff m_cfg.m_cpol == 0);
12       req_data[i] = vif.mosi;
13     end
14
```

```
15      case(spi_cmd::convert_to_cmd_e(req_data))
16        SPI_WRITE: begin
17          m_item = spi_write_cmd::type_id::create();
18          req_data = new[48](req_data);
19        end
20        SPI_READ: begin
21          m_item = spi_read_cmd::type_id::create();
22          req_data = new[32](req_data);
23          rsp_data = new[16];
24        end
25        SPI_STATUS: begin
26          m_item = spi_status_cmd::type_id::create();
27          rsp_data = new[48];
28        end
29      endcase
30
31      //read remaining request
32      for (int i = 16; i < req_data.size(); i++) begin
33        @(posedge vif.sclk iff m_cfg.m_cpol == 1
34          or negedge vif.sclk iff m_cfg.m_cpol == 0);
35        req_data[i] = vif.mosi;
36      end
37      //read response
38      foreach(rsp_data[i]) begin
39        @(posedge vif.sclk iff m_cfg.m_cpol == 1
40          or negedge vif.sclk iff m_cfg.m_cpol == 0);
41        rsp_data[i] = vif.miso;
42      end
43
44      m_item.set_req_data(req_data);
45      m_item.set_rsp_data(rsp_data);
46      ap.write(m_item);
47    end
48  endtask : run_phase
```

Listing 7.7: SPI monitor, run phase task (monitoring)

## 7.4 Scoreboard

An automatic validation is indispensable for exhaustive constraint-random testing. Therefore, the scoreboard is one of the major classes in the Test Bench. It holds a model of the tag and validates the responses of the tag by using it.

The scoreboard, or basically the model, is configured by a configuration object called `cts_sb_config`. It is needed to define the model tags memory. There are attributes to describe the size for each memory block of the EPC memory. It is also possible to initialize the memory with a file. The path to the file has to be set in the configuration object. As well the start addresses of each block inside the file has to be defined. The start address and the memory size are defined in multiples of words.

Additionally, the scoreboard holds three coverage groups to allow the calculation of functional coverage.

- `cmds_cg`: This group covers which commands were used during the test run.
- `states_cg`: covers which states were visited during the test run
- `state_trans_cg`: This group records which possible state transitions were performed. Figure 7.4 shows which transitions are allowed.

Every check starts in either the `write_EPC` or the `write_SPI` method. This methods are called by the corresponding monitor when writing a sequence item to the analysis port. In this methods the items are delegated to the appropriate check method. In the check method the response of the command is verified and the model is updated. Afterwards the functional coverage groups are sampled.

The check methods are basically large if-else trees. Figure 7.3 shows such a tree for the write command. Arrows out of the bottom edge of a diamond are to follow in case the condition is true. Out of the side edges it is for the case the condition is false. If there is more than one condition or process following, they are executed in order from left to right. As it can be seen most of the processes print error messages. This is done by using the `` `uvm_error(ID,msg) `` or the `` `uvm_fatal(ID,msg) `` macro. They print the error in a special format. The error macro prints the message and counts the errors grouped by ID. The fatal macro prints the message and ends the run. A fatal is used if the model can not be set to a valid state because of the error. In the example the `` `uvm_fatal `` macro is used if the data should be saved into the model.

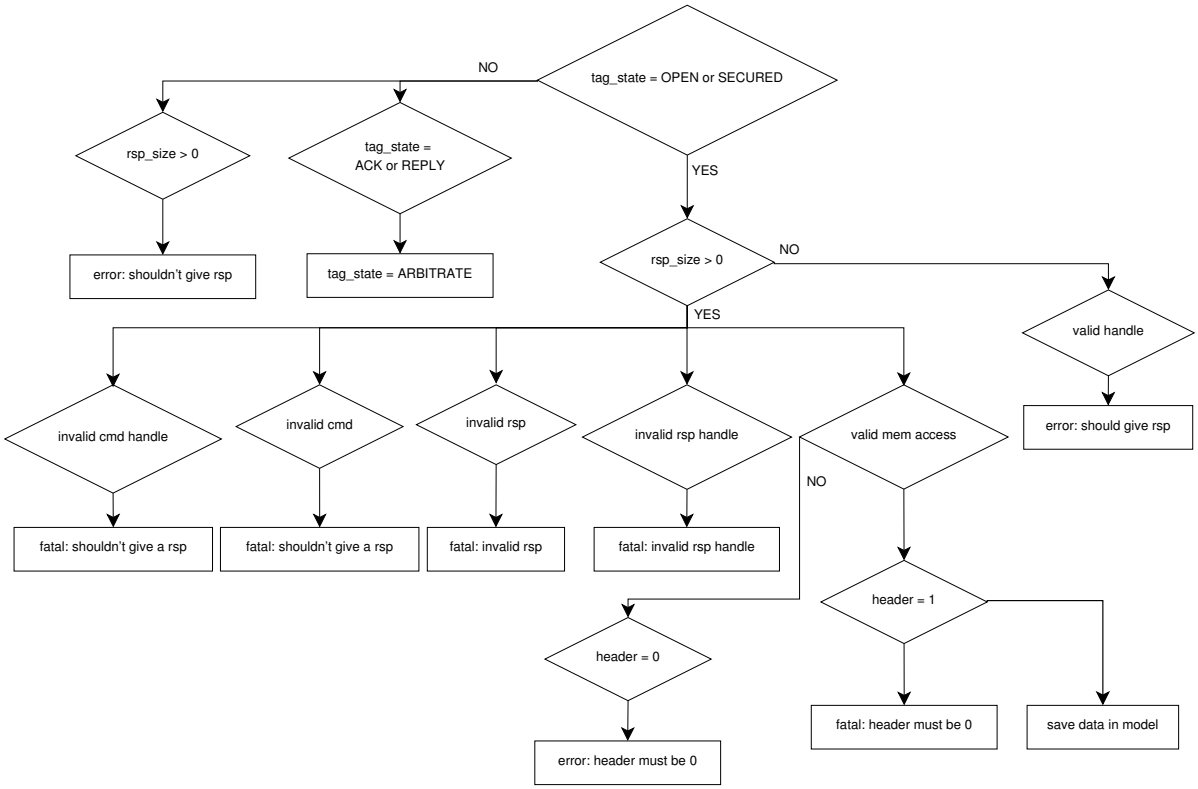


Figure 7.3: If-else tree of the check write method

## 7.5 Sequences

Sequences generates sequence items and runs them on a sequencer. Besides they also run sub-sequences, which then run other sub-sequences or sequence items. The Test Bench offers several sequences which can be used to generate stimuli.

`epc_base_seq` is a base class for all sequences concerning the EPC interface. Therefore all such sequences are subclasses of `epc_base_seq`. The class holds a reference to the `cts_sb_config` configuration object as well as one to `epc_agent_config`. Furthermore the `pre_start()` task is implemented to get the configuration objects from the library.

`epc_rand_cmd_seq` can be used to run a random EPC command. By default the sequence chooses one out of all implemented commands. Obviously, by constraining the member variable `cmd` the amount of possible commands can be reduced.

`epc_query_adjust_rep_seq` either runs a QUERY_REP or a QUERY_ADJUST. The sequence has three random variables. The command variable is constrained in a way that three out of four commands will be a QUERY_REP. The other variables are `session` and `up_dn` which is only for QUERY_ADJUST. The possible responded $RN16$ is saved into the variable `rsp_rn16`.

`epc_do_query_seq` only runs a QUERY and catches the response which is then saved into `rsp_rn16`. This makes life a bit easier when running a QUERY. The sequence has a random variable for every field of the command.

`epc_set_open_seq` sets the tag into the OPEN state. First a QUERY is executed. There are random variables for the fields *sel*, *session* and *target*, as well as the variable `q_start` which defines the $Q$ field. Afterwards a `epc_query_adjust_rep_seq` is executed as long as the tag stays in the ARBITRATE state or a maximum count is reached (defined over the random variable `max_count`). If the maximum count was reached, the sequence runs QUERY_ADJUST commands with the purpose to decrease $Q$ until the tag answers. Next an ACK and afterwards a REQ_RN command is executed to set the tag into the OPEN state. In the end the variable `rsp_rn16` holds the responded $RN16$ and `handle` the handle.

`epc_rand_rw_seq` can be used to run a random READ or WRITE command. The sequence has three random variables, `cmd` which can be one of the two commands, `handle` and `mem_bank`. To be sure that the commands do a memory access, the handle have to be constrained to be the right one, at the sequence call. The *word_ptr* is constraint depending on the *mem_bank*. In case of a READ, the *word_count* is constrained depending on the before calculated *word_ptr* and the size of the memory bank.

`epc_access_seq` runs the two staged access procedure. The random variable `num_cmds_between` defines the amount of random commands which should be executed between the two ACCESS commands. The other two random members defines the *handle* and the access password. Usually they are constrained at sequence call.

`epc_kill_seq` runs the two staged kill procedure. Understandably it is analog to the `epc_access_seq`.

`spi_base_seq` is the base class for sequences using the SPI interface. It holds a reference to the `spi_agent_config` configuration object which is loaded during the *pre_start* phase.

`cts_base_vseq` is the base class for sequences serving both interfaces. In general these will be virtual sequences which means that they are only running sub-sequences.

### 7.5.1 Set to a state sequence library

This library offers the possibility to set the tag from the current state to a possible successor state. An appropriate randomly chosen command is executed to reach the desired state. Figure 7.4 shows all possible state transitions and also which commands will lead to this change. In Table 7.1 the commands are listed. Also the conditions which are needed to achieve the desired result are notated. As an example to get from READY to ARBITRATE a QUERY with matching flags and a slot counter not equal to zero is needed. To reach the REPLY state the slot counter must be zero. This can be assured by setting $Q$ to zero. All other (number 90) in the table means every other command which is not directly mentioned in this group of transitions.

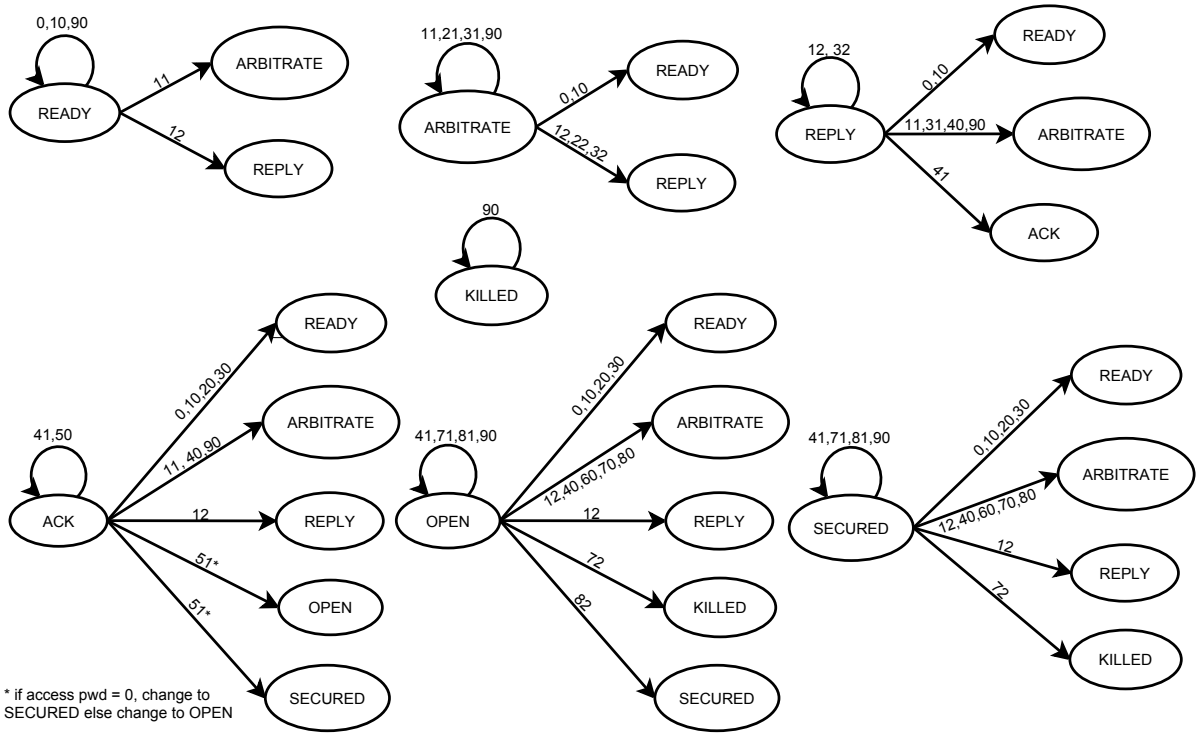


Figure 7.4: All possible state transitions

| num | command | condition |
|---|---|---|
| 0 | SELECT | |
| 10 | QUERY | flags not matching |
| 11 | QUERY | matching flags and slot != 0 |
| 12 | QUERY | matching flags and slot == 0 |
| 20 | QUERY_REP | |
| 21 | QUERY_REP | valid session and slot != 0 or invalid session |
| 22 | QUERY_REP | valid session and slot == 0 |
| 30 | QUERY_ADJUST | |
| 31 | QUERY_ADJUST | valid session and slot != 0 or invalid session |
| 32 | QUERY_ADJUST | slot == 0 and valid session |
| 40 | ACK | invalid RN16 |
| 41 | ACK | valid RN16 |
| 50 | REQ_RN | invalid RN16 |
| 51 | REQ_RN | valid RN16 |
| 60 | NAK | |
| 70 | KILL | valid handle and invalid kill pwd |
| 71 | KILL | invalid handle |
| 72 | KILL | valid handle and valid kill pwd |
| 80 | ACCESS | valid handle and invalid access pwd |
| 81 | ACCESS | invalid handle |
| 82 | ACCESS | valid handle and valid access pwd |
| 90 | all other | |

Table 7.1: Commands for state transitions

There is a sequence for every origin state. For example the `cts_from_open_vseq` is in charge of setting the tag from OPEN to a follow-up state. Listing 7.8 shows the randomization part of the sequence. At first, `desired_state` is constrained to the possible amount of successor states. The idea behind having a non-uniform distribution is to have a higher probability to reach not easily accessible states. SECURED for example is only reachable from OPEN and under certain conditions from ACK. Next, depending on the selected state the command which should be run is constrained. Note that *RESET* is used as a placeholder for random other command (number 90 in the table). If it is needed the distributions can be weighted other than equally.

```
1  rand tag_state_e desired_state;
2  rand epc_cmds_e next_cmd;
3  rand logic[0:1] desired_session;
4
5  rand logic[0:15] rn16;
6  logic[0:15] returned_rn16;
7  logic[0:15] actual_session;
8
9  constraint follow_up_states {desired_state dist {READY := 2,
10   ARBITRATE := 2, REPLY := 2, OPEN := 3, SECURED := 4, KILLED := 2};}
11 //RESET is a placeholder for random other cmd
12 constraint possible_cmds {
13   desired_state == READY -> next_cmd dist {SELECT := 1, QUERY := 1,
        QUERY_REP := 1, QUERY_ADJUST := 1};
14   desired_state == ARBITRATE -> next_cmd dist {QUERY := 1, ACK := 1,
        NAK := 1, KILL := 1, ACCESS := 1};
15   desired_state == REPLY -> next_cmd == QUERY;
16   desired_state == OPEN -> next_cmd dist {KILL := 1, ACCESS := 1, ACK
        := 1, RESET := 1};
17   desired_state == SECURED -> next_cmd == ACCESS;
18   desired_state == KILLED -> next_cmd == KILL;
19 }
20 constraint var_order {solve desired_state before next_cmd;}
```

Listing 7.8: Random variables and constraints of `cts_from_open_vseq`

The `body()` task of the sequence only delegates the work to an appropriate task, depending on the desired successor state. Listing 7.9 demonstrates such a task. The purpose of this task is to keep the tag in the OPEN state. As it can also be seen in the listing above the possible commands are KILL, ACCESS, ACK and a random other command. For the KILL and ACCESS commands, or better the correspondent procedure, the particular sequence is used. To match the condition (see also Table 7.1) the handles are constrained to be invalid. The ACK command is run with a valid *RN*16. The RESET case is the all other case. This means one out of all not explicitly in a "OPEN to X" transition used commands.

```
virtual protected task to_open();
  epc_kill_seq kill;
  epc_access_seq access;
  epc_rand_cmd_seq rand_cmd;
  epc_ack_cmd ack;

  case(next_cmd)
    KILL:
      `uvm_do_on_with(kill, epc_sqr,
      {handle != rn16;
       num_cmds_between == 0;
      })
    ACCESS:
      `uvm_do_on_with(access, epc_sqr,
      {handle != rn16;
       num_cmds_between == 0;
      })
    ACK:
      `uvm_do_on_with(ack, epc_sqr,
      {ack.m_rn16_cmd == rn16;})
    RESET:
      `uvm_do_on_with(rand_cmd, epc_sqr,
      {cmd != RESET && cmd != SELECT && cmd != QUERY && cmd !=
          QUERY_REP
       && cmd != QUERY_ADJUST && cmd != NAK && cmd != KILL && cmd !=
           ACCESS && cmd != ACK;})
  endcase
endtask: to_open
```

Listing 7.9: Task to run a command to set the tag OPEN

However, there is the `cts_set_state_vseq` sequence which offers the simplest and recommended way to use this library. This sequence is not used like a normal sequence. Listing 7.10 demonstrates a short how-to demo. The `` `uvm_do() `` macro is only used to initialize the sequence because the `body()` task does nothing. Anyhow, constraints for the two random variables `session` and `rn16_handle` can be defined. This variables defines the session parameter for QUERY commands and the *rn* or *handle* parameter for appropriate commands. The `set_to_state(desired_state)` is the most interesting task. The only parameter defines the desired target state. First the task gets the current state from the scoreboard and afterwards it calls the appropriate from_x sequence. If it is not possible to reach the desired state from the current one an error message will be printed. Actually the error message is that the randomization failed in the from_x sequence. Besides no command will be executed. By calling `set_to_state(RAND)` the successor state is chosen randomly.

```
cts_set_state_vseq set_state;
`uvm_do(set_state)
set_state.set_to_state(REPLY);
set_state.set_to_state(RAND);
```

Listing 7.10: How to use `cts_set_state_vseq`

## 7.6 Tests

The tests are basically implemented as it is described in section 5.3.11. There are three base classes defined. `epc_base_test` for tests using the EPC interface, `spi_base_test` in case the SPI interface is used and `cts_base_test` if both interfaces are needed. As an example Listing 5.9 shows the base class and a derived class for SPI.

## 7.7 Implementation

The Test Bench was implemented with the help of the Eclipse SDK and the Design and Verification Tools (DVT) plug-in. Compiling and simulation were done with Mentor Graphics Questa Sim 10.0c. The UVM is supported since version 10.0a.

The simulator also provides functional coverage reports. Figure 7.5 shows such a report. The three defined coverage groups, `state_trans_cg`, `states_cg` and `cmds_cg` and how much each of them is covered is presented. Furthermore the coverage points and their coverage is shown, as well as the corresponding bins. As an example, it can be seen that the tag first switched from the READY to the REPLY state, next to ARBITRATE and afterwards it stays there.

| Name | Coverage | Goal | % of Goal | Status |
|---|---|---|---|---|
| /cts_pkg/cts_scoreboard | | | | |
| TYPE state_trans_cg | 13.0% | 100 | 13.0% | |
| INST \/cts_pkg::cts_scoreboard::st... | 13.0% | 100 | 13.0% | |
| CVP ready_cp | 33.3% | 100 | 33.3% | |
| bin ready[READY=>REPLY] | 1 | 1 | 100.0% | |
| bin ready[READY=>ARBITRATE] | 0 | 1 | 0.0% | |
| bin ready[READY=>READY] | 0 | 1 | 0.0% | |
| CVP arbitrate_cp | 33.3% | 100 | 33.3% | |
| bin arbitrate[ARBITRATE=>REPL... | 0 | 1 | 0.0% | |
| bin arbitrate[ARBITRATE=>ARBI... | 1 | 1 | 100.0% | |
| bin arbitrate[ARBITRATE=>READ... | 0 | 1 | 0.0% | |
| CVP reply_cp | 25.0% | 100 | 25.0% | |
| bin reply[REPLY=>ACKNOWLEDGED... | 0 | 1 | 0.0% | |
| bin reply[REPLY=>REPLY] | 0 | 1 | 0.0% | |
| bin reply[REPLY=>ARBITRATE] | 1 | 1 | 100.0% | |
| bin reply[REPLY=>READY] | 0 | 1 | 0.0% | |
| CVP ack_cp | 0.0% | 100 | 0.0% | |
| CVP open_cp | 0.0% | 100 | 0.0% | |
| CVP secured_cp | 0.0% | 100 | 0.0% | |
| CVP killed_cp | 0.0% | 100 | 0.0% | |
| TYPE states_cg | 42.8% | 100 | 42.8% | |
| INST \/cts_pkg::cts_scoreboard::st... | 42.8% | 100 | 42.8% | |
| CVP states_cp | 42.8% | 100 | 42.8% | |
| ignore_bin ig | 0 | - | - | |
| bin auto[READY] | 1 | 1 | 100.0% | |
| bin auto[ARBITRATE] | 2 | 1 | 100.0% | |
| bin auto[REPLY] | 1 | 1 | 100.0% | |
| bin auto[ACKNOWLEDGED] | 0 | 1 | 0.0% | |
| bin auto[OPEN] | 0 | 1 | 0.0% | |
| bin auto[SECURED] | 0 | 1 | 0.0% | |
| bin auto[KILLED] | 0 | 1 | 0.0% | |
| TYPE cmds_cg | 65.3% | 100 | 65.3% | |
| INST \/cts_pkg::cts_scoreboard::cm... | 65.3% | 100 | 65.3% | |
| CVP cmds_epc_cp | 30.7% | 100 | 30.7% | |
| ignore_bin ig | 0 | - | - | |
| bin auto[QUERY_REP] | 0 | 1 | 0.0% | |
| bin auto[ACK] | 1 | 1 | 100.0% | |
| bin auto[QUERY] | 1 | 1 | 100.0% | |
| bin auto[QUERY_ADJUST] | 0 | 1 | 0.0% | |
| bin auto[SELECT] | 0 | 1 | 0.0% | |
| bin auto[NAK] | 0 | 1 | 0.0% | |
| bin auto[REQ_RN] | 5 | 1 | 100.0% | |
| bin auto[READ] | 0 | 1 | 0.0% | |
| bin auto[WRITE] | 0 | 1 | 0.0% | |
| bin auto[KILL] | 0 | 1 | 0.0% | |
| bin auto[LOCK] | 0 | 1 | 0.0% | |
| bin auto[ACCESS] | 0 | 1 | 0.0% | |
| bin auto[RESET] | 1 | 1 | 100.0% | |
| CVP cmds_spi_cp | 100.0% | 100 | 100.0% | |
| bin auto[SPI_WRITE] | 1 | 1 | 100.0% | |
| bin auto[SPI_READ] | 2 | 1 | 100.0% | |
| bin auto[SPI_STATUS] | 1 | 1 | 100.0% | |

Figure 7.5: Functional coverage report of Questa Sim

# 8 Conclusions

This chapter concludes the thesis. It gives a summary over the work and the results. Besides some words about further work are noted.

## 8.1 Summary and Results

The aim of this thesis was to develop a coverage driven constraint random verification environment with UVM to verify a combined passive HF/UHF RFID tag. To do so the Test Bench should communicate with the EPC interface as well as with the SPI interface.

First an introduction over RFID systems was given. Furthermore the EPC standards for UHF as well as for HF were described in more detail. Also the SPI bus was explained as well as the protocol used by the interface of the DUT. A significant part is occupied by an introduction to SystemVerilog. The most important enhancements to Verilog as well as the new features, like classes or constructs for functional coverage and constraint randomization, were described. One chapter is dedicated to UVM. In it the concept as well as the individual components are explained. A prominent part of the thesis is devoted to the developed Test Bench.

The developed verification environment meets the requirements. It can be extended easily. For example, a new agent for the NFC interface can be easily integrated into the Test Bench by adding a correspondent agent to the environment and adding testing methods to the scoreboard. By following the concept of UVM the individual components are reusable. Through the configuration objects it is a well configurable Test Bench. The sequence items (test vectors) can be generated with random values. Secondary constraints can be defined to limit the possible values. Besides some coverage groups are defined to obtain functional coverage.

In comparison to the old test environment, the new one allows randomized and exhaustive testing. A big advantage is also that the new Test Bench is self checking, whereas with the old one the results have to be checked manually.

In general it can be said, that the UVM is a powerful verification tool which offers many options to create a rich verification environment. The creation of the Test Bench may cost a little bit more effort but the results compensates it easily.

## 8.2 Further Work

A big chunk is to extend the Test Bench to be able to verify the NFC interface. Another expansion could be to add some more coverage groups, for example inside of the commands to receive more specific functional coverage. Besides, because of some reasons, the EPC driver doesn't fully support the HF version 2.0.3 protocol. Therefore the driver should be updated.

# Appendix

# Bibliography


[1] Accellera systems initiativey. http://www.accellera.org.

[2] All about fork-join of system verilog. http://learn-systemverilog.blogspot.de/2009/08/all-about-fork-join-of-system-verilog.html.

[3] systemverilog.in. http://www.systemverilog.in.

[4] Testbench.in. http://www.testbench.in.

[5] Accellera. *SystemVerilog 3.1a Language Reference Manual*, 2004.

[6] Accellera. *Universal Verification Methodology (UVM) 1.1 Class Reference*, June 2011.

[7] Accellera. *Universal Verification Methodology (UVM) 1.1 User's Guide*, May 2011.

[8] Thomas Andrejka. Msc. Master's thesis, FH Hagenberg, June 2007.

[9] Mark Glasser. *Open Verification Methodology Cookbook*. Springer, 1st edition. edition, 7 2009.

[10] Mentor Graphics. Verification academy. http://verificationacademy.com.

[11] GS1 EPCglobal. *EPC Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz - 960 MHz Version 1.2.0*, October 2008.

[12] GS1 EPCglobal. *EPC Radio-Frequency Identity Protocols Class-1 HF RFID Protocol for Communications at 13.56 MHz Version 2.0.3*, September 2011.

[13] Johann Heyszl. Msc. Master's thesis, University of Technology Graz, July 2007.

[14] Infineon Technologies. *CRE SPI Interface Specification*, 2012. unpublished.

[15] Wikipedia. Radio-frequency identification — wikipedia, the free encyclopedia, 2012. [Online; accessed 12-June-2012].

[16] Wikipedia. Serial peripheral interface bus — wikipedia, the free encyclopedia, 2012. [Online; accessed 11-June-2012].

[17] Wikipedia. Systemverilog — wikipedia, the free encyclopedia, 2012. [Online; accessed 18-June-2012].