Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Microelectronics

Verification component implementation - Model of NFC

Bc. Ján Jendrušák

Supervisor: doc. Ing. Jiří Jakovenko, Ph.D.
Field of study: Electronics and Communications
Subfield: Electronics
May 2019

MASTER'S THESIS ASSIGNMENT

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

I. Personal and study details

Student's name:	**Jendrušák Ján**	Personal ID number:	**475404**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Microelectronics**		
Study program:	**Electronics and Communications**		
Branch of study:	**Electronics**		

II. Master's thesis details

Master's thesis title in English:

Verification Component Implementation - Model of NFC

Master's thesis title in Czech:

Implementace verifikační komponenty - modelu NFC přijímače

Guidelines:

1. Study NFC (Near Field Communication) Type 2 Tag Platform protocol according to [1] and techniques of verification component (UVC - Universal Verification Component) implementation based on UVM (UV Methodology) framework.
2. Implement NFC Forum Device Type 2 UVC supporting NFC protocol according to specification [1](chap.2.1,3,4 and 5) This UVC shall therefore:
- Support the commands from [1] specification: ALL_REQ, SENS_REQ, SELL_REQ, SDD_REQ, SLP_REQ, READ, WRITE, SECTOR SELECT
- Identify all possible responses to previously stated commands
- Model the Static Memory Structure and NFC Type 2 Tag Platform State Machine
- Validate the timing between the response of DUT and command from UVC and compare actual response with the expected response
3. Write verification plan for NFC Type 2 Tag Platform based on [1] and add coverage points to cover the requirements.
4. Demonstrate UVC functionality by writing an example of a constrained-random test. As a DUT use reference NFC Type 2 Tag platform design module from ASICentrum.

Bibliography / sources:

[1] NFCForum. Type 2 Tag Operation Specification. Verze 1.1, 2011-05-31.

Name and workplace of master's thesis supervisor:

doc. Ing. Jiří Jakovenko, Ph.D., Department of Microelectronics, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **07.02.2019** Deadline for master's thesis submission: 24.05.2019

Assignment valid until: **20.09.2020**

[illegible]	[illegible]	[illegible]
doc. Ing. Jiří Jakovenko, Ph.D.	Head of department's signature	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature		Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

7.3.2019 — Date of assignment receipt

[illegible] — Student's signature

Acknowledgements

I would like to thank my supervisor doc. Ing. Jiří Jakovenko, Ph.D. for his guidance and useful advice everytime I needed one. I would also like to thank ASICentrum company for the great opportunity to work on this master's thesis for their team, especially Ing. Luboš Hradecký for arranging the whole cooperation, Ing. Martin Jäger and Bc. Henrique Hellini for consultations about the testbench development, and Ing. Jan Bičák, Ph.D. and Ing. Tomáš Novák for help with the NFC Type 2 Tag Platform specification and the DUT incorporation into the UVC.

Declaration

I hereby declare that the presented work was carried out independently and that I have listed all information sources used in accordance with the Methodical Guidelines on Maintaining Ethical Principles During the Preparation of Higher Education Theses. Furthermore, I declare that the borrowing and publishing of my thesis or part of it is allowed with the agreement of department.

In Prague, 22. May 2019

Abstract

To design digital circuits is a complex process, which involves verification of design behaviour before it is actually implemented. Universal Verification Methodology (UVM) promises an efficient way of verifying digital designs with any major simulation platform.

Using UVM is demonstrated in this paper to build a verification component for NFC Type 2 Tag Platform, the operational specification for wireless tags in high frequency band. Completely functioning UVC together with prepared verification plan ensures that the Device Under Test (DUT) complies with NFC Type 2 Tag Platform on logical level.

Keywords: verification, Universal Verification Methodology, Near Field Communication, SystemVerilog, tag

Supervisor: doc. Ing. Jiří Jakovenko, Ph.D.

Abstrakt

Návrh digitálnych obvodov je komplexný proces, ktorý zahŕňa verifikáciu chovania návrhu predtým, ako je na záver implementovaný. Universal Verification Methodology (UVM) je prísľubom efektívneho spôsobu verifikovania digitálnych návrhov s ktoroukoľvek simulačnou platformou.

Použitie UVM je demonštrované v tejto diplomovej práci návrhom verifikačného komponentu pre NFC Type 2 Tag Platform, špecifikácie pre bezdrôtové tagy vo vysokom frekvenčnom pásme. Kompletne funkčný verifikačný komponent spolu s predpripraveným verifikačným plánom by malo zaručiť, že digitálny návrh splňuje požiadavky protokolu na logickej úrovni.

Kľúčové slová: verifikácia, Universal Verification Methodology, Near Field Communication, SystemVerilog, tag

Preklad názvu: Implementácia verifikačného komponentu – modelu NFC prijímača

Contents

Figures

Tables

0.1 List of Abbreviations

ACK	Acknowledgement
ASIC	Application Specific Integrated Circuit
ASK	Amplitude Shift Keying
CC	Capability Container
CRC	Cyclic Redundancy Check
CDV	Coverage Driven Verification
DUT	Device Under Test
EoF	End of Frame
FDT	Frame Delay Time
GUI	Graphics User Interface
ISO	International Organization for Standardization
MDV	Metric Driven Verification
NACK	Negative Acknowledgement
NDEF	NFC Data Exchange Format
NFC	Near Field Communication
FPGA	Field Programmable Gate Array
OOK	On/Off Keying
RFU	Reserved for Future Use
RTL	Register Transfer Level
SoF	Start of Frame
TB	Testbench
UVC	Universal Verification Component
UVM	Universal Verification Methodology

Chapter 1

Introduction

Designing digital circuits is a process with many steps on different level of abstraction. In the beginning the whole scope of work must be defined. Project team has to choose a target technology - whether Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC) will be used. Also other aspects of the project are specified, like required features, way of their implementation, supply voltage, design interface and other factors.

After the specification is complete, a team of digital designers creates Register Transfer Level (RTL) code according to the specification, mostly in VHDL or Verilog hardware description language. The correct behaviour of the design should be verified from the beginning of RTL code creation in order to detect as many bugs as possible. To achieve this, RTL design is inserted into testbench environment, where the set of various test cases is performed.

Latest major verification methodology used for digital designs is called Universal Verification Methodology (UVM), which should be suitable for every type of digital circuit. Using UVM promises efficient way of verifying digital designs independently of simulation platform.

This master's thesis describes all important steps during the design of complete verification solution in cooperation with ASICentrum company, design center of EM Microelectronic and part of the Swatch Group. EM Microelectronic focuses on ultra-low power ASICs for industry use, traceability, logictics and wearables. One of the implemented wireless communication protocols is NFC, namely NFC Type 2 Tag Platform specification, and this paper deals with this particular protocol.

The main goal is to develop Universal Verification Component (UVC) for

NFC Type 2 Tag Platform using UVM library. That way the verification component will be compliant with the latest ASICentrum verification flow and could be inserted into new testbenches to verify new digital designs. Besides that, the understanding of technical documentation together with practical UVC implementation is a great practice for the future verification engineer position.

Created UVC will be evaluated by testing already designed and verified NFC Type 2 Tag logic core from ASICentrum. In order to fullfill this task, all of the necessary information is gathered, therefore the first two chapters cover teoretical background of given task - characteristics of NFC Type 2 Tag Platform and UVM library fundamentals.

The next chapters describe the proposed verification plan for the NFC Type 2 Tag Platform, created verification environment and each UVM component inside the environment. UVC functionality is then demonstrated with ASICentrum logic core with the set of constrained-random written test cases and designed UVC is evaluated.

Part I

Theoretical Part

Chapter 2

Near Field Communication

Wireless comunnication is a major technological trend mainly due to the ability to enable more comfortable connections between electronic devices. One of already well established technologies is Near Field Communication (NFC). NFC is promoted by non-profit association - The NFC Forum, and provides short-range and secure comunnication between NFC Devices, which may operate in 3 modes [3] [4].

The first mode is called Card emulation mode. NFC Device in Card emulation mode acts as a smart card, thus permitting payments for services and goods through communication with an external reader [4]. Another mode is Peer-to-Peer Mode, enabling NFC devices to exchange various data, for example contact information or WiFi passwords. The last mode's name is Reader/Writer Mode, in which one NFC Device's (labeled as reader) role is to read information from the passive NFC Devices - NFC Tags [3].

NFC Forum specifies 5 different types of NFC Tags, each of them is suited for a specific purpose and price range [5]. Main goal of this thesis is to create verification component for Type 2 Tag Platform, therefore only this NFC Forum Platform is described in more detail .

2.1 Type 2 Tag Platform

Type 2 Tag uses a particular technology subset of NFC-A Technology including anticollision and it is based on ISO 14443-3 A standard. [6] The device that starts the communication - NFC Forum Device in Poll Mode, transmits elecromagnetic field of frequency 13.56 MHz (f_c). NFC Tag (also named NFC Forum Device in Listen Mode) detects this field and responds to NFC

Commands according to its own internal state. Both the reader and the tag communicate with each other using the same bitrate 106 $kbaud/s$ and the period of one bit is therefore [1]

$$per_{bit} = \frac{128}{f_c} = \frac{128}{13.56\ MHz} \cong 9.44\ \mu s.$$

2.1.1 Memory structure

Type 2 Tag memory structure depends on the memory size of the tag. If the memory size is equal to 64 bytes, a static memory structure is used. Otherwise, if the memory size is bigger than 64 bytes, a dynamic memory structure is appplied to Tag memory.

Both structures are divided into blocks and each block contains 4 bytes. Static and dynamic structure also share basic memory fields:

- *Internal*: reserved for manufacturer use
- *Lock*: static lock bytes, their purpose is to switch between READ/WRITE and READ-ONLY States
- *CC*: Capability Container, manages the information of the Type 2 Tag Platform
- *Data*: available area for information storage. TLV blocks may be stored here - block of 1 to 3 fields. One byte T field specifies a type of TLV, one or three bytes long L field defines the size in bytes of the value field, and V field may store values. If the L field is equal to 00h or there is no L field, the V field is not present.

Dynamic memory structure also has dynamic lock bytes and reserved bytes, located between or at the end of the data. Complete description of the memory structure can be found in [2].

2.1.2 Sequence schemes

NFC Reader issues a command to NFC Tag by pausing the carrier transmission for a defined time $t_1 = 2 - 3.4\mu s$, also called V_2. This way 3 particular patterns are created, (see Figure 2.1). Pattern X has V_2 after the half duration of one bit, pattern Y has no modulation throughout the bit duration and in pattern

Z V_2 must occur at the beginning of current bit. This mechanism is known as Modified Miller coding with Amplitude Shift Keying (ASK) 100 % [1] [7].

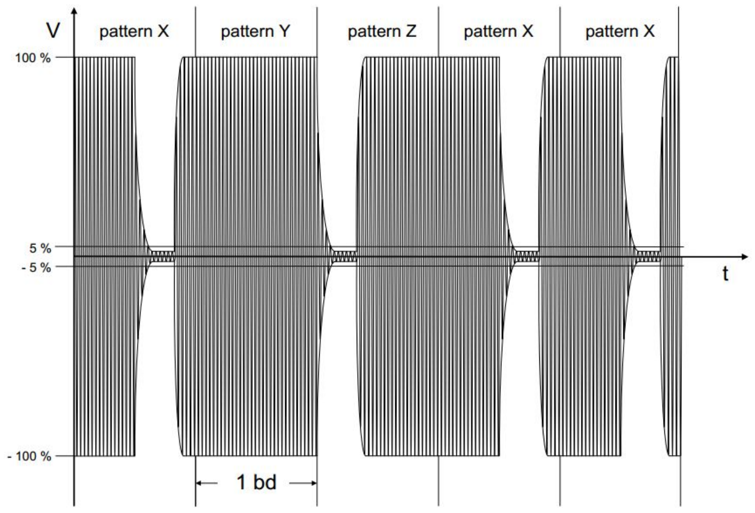


Figure 2.1: Modified Miller with 100 % ASK [1].

NFC Tag - Device in Listen mode, modulates the analog signal by Manchester coding with On/Off Keying (OOK) subcarrier modulation (see Figure 2.2). With OOK modulating subcarrier (subcarrier frequency $f_c/16$) 3 patterns are defined - D, E and F. In pattern D the subcarrier modulates the carrier for the first half of the bit duration, while in the second half modulation must not occur. Pattern E has a modulated carrier by the subcarrier for the second half of the bit duration and pattern F has no subcarrier modulation during its whole bit duration [1].

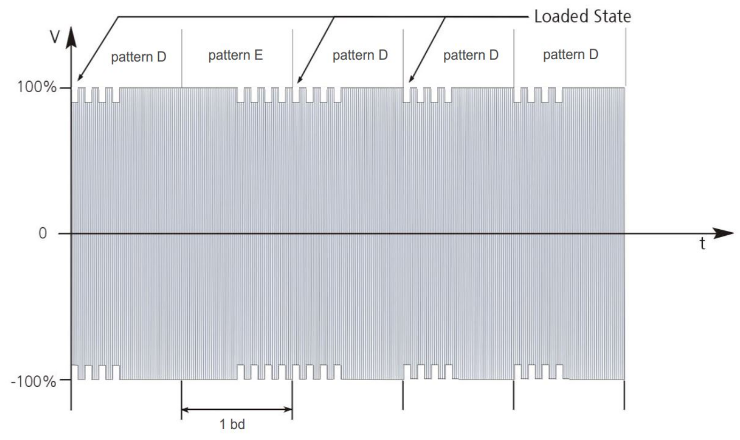


Figure 2.2: Manchester coding with OOK [1].

2.1.3 Bit level coding

Patterns mentioned in Section 2.1.2 are used to code two logical levels - logic 0 and logic 1. NFC Forum Device uses these patterns as it is stated here

pattern X logic 1,

pattern Y logic 0,

pattern Z during two or more contiguous logic 0, pattern *Z* is used from the second logic 0, pattern *Z* is also used as Start of Frame (SoF, see 2.1.4) [1].

NFC Type 2 Tag codes his response as follows:

pattern D logic 1, SoF

pattern E logic 0 [1].

2.1.4 Frame format

Commands and responses of NFC Devices are encapsulated within the frames, which group data bits together with addition of SoF and End of Frame (EoF). NFC-A Technology defines three types of frames - short frame, standard frame, and bit-oriented SDD frame [1] [8].

The short frame is used for initiating the communication. It consists of SoF, maximum of 7 data bits, and EoF. The short frame does not use parity protection [1].

The standard frame sends more than one data bytes and adds odd parity protection at the end of each one. Data bytes are limited from the start with SoF and with EoF at the end [1].

The bit-oriented SDD frame is used for solving collision. It is derived from the standard frame with 7 data bytes split up into 2 parts. Part 1 is sent by NFC Device in Poll Mode and part 2 by NFC device in Listen Mode. Complete description of this frame is located in [1].

2.1.5 Commands and responses

Table 2.1 lists all the commands available for NFC Forum Type Tag 2 Platform. For each command additional information are stated - command code and the frame type.

NFC Type 2 Command	**Command code**	**Response**	**Frame Type cmd/res**
SENS_REQ	26h	SENS_RES	short/standard
ALL_REQ	52h	SENS_RES	short/standard
SEL_REQ	9xh	SEL_RES	standard
SDD_REQ	9xh	SDD_RES	bit oriented
SLEEP_REQ	50h 00h	-	standard
READ	30h	READ/NACK response	standard
WRITE	A2h	ACK/NACK response	standard
SECTOR SELECT	C2h	ACK/passive ACK/NACK	standard

Table 2.1: NFC Type 2 Commands [1] [2]

SENS_REQ and *ALL_REQ* commands are used for detecting NFC Forum Type 2 Devices in Listen Mode. Difference between these commands is Type 2 Tag responds to *ALL_REQ* in *SLEEP_A* state (see Figure 4.1) [1] [6].

SDD_REQ is used for resolving the collision (whether more than one device is in Operating Field of NFC Forum Device in Poll Mode). *SDD_REQ* also serves for obtaining NFCID1 of an NFC Forum Device in Listen Mode [1] [6].

SEL_REQ selects the NFC Forum Device in Listen Mode according to sent NFCID1 of length 4 bytes. These 4 bytes are additionally protected with BCC byte - calculated as exclusive-or checksum [1] [6].

Issued *SLEEP_REQ* command puts NFC Forum Device in Listen Mode to *SLEEP_A* state. NFC Forum Device in Listen Mode must not respond to this command and NFC Forum Device in Poll mode must always consider this command to be executed properly [1] [6].

READ command consists of the command code 30h and one parameter - block number (*BNo*). Response from the NFC Type 2 Tag Platform is 16 bytes long payload of data, which means 4 concurrent blocks are returned to NFC Forum Device in Poll Mode. In case of error, Type 2 Tag Platform responds with $NACK$ response [2].

NFC Forum Device in Poll Mode may also write data to Type 2 Tag Platform by issuing $WRITE$ command. Only the whole block is written with one $WRITE$ command,. If the $WRITE$ command is executed successfully, the Acknowledgement (ACK) response is returned to NFC Device in Poll Mode. Otherwise, the Negative Acknowledgement ($NACK$) response is returned [2].

The last command in command set of this platform is $SECTOR\ SELECT$, which is designated for physical memories bigger than 1 KB - is used to switch between 1 KB big sectors. The $SECTOR\ SELECT$ command is issued with 2 separated packets. The goal of the first packet is to get information whether memory is actually bigger than 1 KB. In that case ACK response is issued by the Type 2 Tag Platform. If the memory is smaller, $NACK$ response is returned. After ACK response the second packet is transmitted with specific sector number and three Reservedf for Future Use (RFU) bytes (00h). If the sector number is inside the available memory, no further response is issued - this is called *passive* ACK Response. If the sector number is outside of the available memory, $NACK$ response is sent back to NFC Forum in Poll Mode [2].

2.1.6 Timing requirements

When it comes to meeting the timing requirements, three important time constants must be fullfilled for NFC Type 2 Tag platform communication. The first one is The Frame Delay Time Poll → Listen, defining the interval in which Listen Frame is allowed to be sent by values $FDT_{A,LISTEN,MIN}$ and $FDT_{A,LISTEN,MAX}$ [1].

If sent command is ALL_REQ, $SENS\_REQ$, SDD_REQ or SEL_REQ, $FDT_{A,LISTEN,MIN}$ equals $FDT_{A,LISTEN,MAX}$ and Listen Frame has to be sent at specific point in time. $FDT_{A,LISTEN,MIN}$ value depends on the logic value of the last data bit - on the position of the last V_2 pulse. $FDT_{A,LISTEN}$ then takes values according to:

$$FDT_{A,LISTEN} = n * 128/f_c + x/fc,$$

where x depends on the last logic bit and n depends on the command, which

Parameter	**Value**
$FDT_{T2T,READ,MAX}$	5 ms
$FDT_{T2T,WRITE,MAX}$	10 ms
$FDT_{T2T,SL,MAX}$	1 ms

Table 2.2: $FDT_{POLL,A,MAX}$ values for *READ*, *WRITE* and *SECTOR SELECT* [1]

is being sent - for previously mentioned groups of commands (ALL_REQ, $SENS\_REQ$, SDD_REQ and SEL_REQ) its value is 9, for all other commands it is equal or bigger than 9 [1]. $FDT_{A,LISTEN}$ must be respected by Tag with tolerance from $-1/f_c$ *to* $0.4\ \mu s + 1/f_c$ [1] [6]. *READ*, *WRITE* and *SECTOR SELECT* commands have their own defined $FDT_{A,LISTEN,MAX}$ values, listed in Table 2.2 [1].

$FDT_{A,LISTEN}$ time also contains two smaller time intervals, both with its own meaning. From the end of the Poll Frame (beginning of the $FDT_{A,LISTEN}$ time), NFC Forum Device must ignore any response from NFC Device in Listen mode during a time $FDT_{A,LISTEN,MIN} - 128/f_c$. Also NFC Device in Listen Mode cannot produce any disturbance before sending a response in time for at least [1]

$$t_{nn,min} = FDT_{A,LISTEN} - (FDT_{A,LISTEN,MIN} - 128/f_c).$$

NFC Forum Device in Poll Mode after the reception of a Listen Frame has to wait certain time, before it sends a new Poll Frame. This time is $FDT_{A,POLL,MIN}$, and it depends on the logic value of the last data bit. That means $FDT_{A,POLL}$ measurement starts from the end of subcarrier's last modulation. This time constant value is not divided into discrete values as $FDT_{A,LISTEN}$ is, it is limited from one side only and the minimum value is $1172/f_c$.

The last requirement is applied to NFC Forum Device in Listen Mode, which must be able to receive initiating commands ALL_REQ or $SENS\_REQ$ after so-called guard time GT_A of unmodulated carrier. Value of this parameter for NFC-A Technology is $5ms$ [1].

2.1.7 NFC Data exchange format - NDEF

NFC Data Exchange Format (NDEF) is lightweight binary message format for encapsulation of one or more payloads with arbitrary size and type into one message construct. NDEF Message is located in Data Area of Memory Structure, more specifically in NDEF Message TLV block (TLV block with 03h T field value, see Section 2.1.1) [9] [10].

Options of managing NDEF Message are obtained by reading Capability Container (CC). The CC is located in the block 3 of the memory structure (both static and dynamic). The CC can also be written with *WRITE* command. Using this command the current content of the CC is bit-wise "OR-ed" with 4 data bytes from *WRITE* command. By this mechanism if a bit is set to logic 1, a change to logic 0 is not possible [2].

Every CC byte has its specific use. Byte 0 should be equal to E1h (so-called magic number), indicating NFC Forum defined data is stored in data area. Byte 1 determines the version number of NFC Type 2 Tag Operation Specification document. Byte 2 specifies memory size in number of 8 bytes and byte 3 indicates read and write access capability of data and CC area [2].

Based on values in the CC and correct presence of NDEF Message TLV in data area (for example L field value is equal to 0), NFC Forum Device might detect the NDEF presence, read the NDEF message using *READ* command or write NDEF message inside NDEF Message TLV with *WRITE* command. If the data to be read or written exceeds the current sector, switch to another sector shall be made with *SECTOR SELECT* command [2].

2.1.8 Anticollision

At first, to be able to handle the situation like the collision, it has to be defined as a phenomenom. Collision happens in case, when more than one NFC Type 2 Tag are responding to *SDD_REQ* command. In that case, the data part of *SDD_RES* response will differ at some point and anticollision loop should be applied. In general concept is simple - sending *SDD_REQ* command with the part of NFCID1 that was already successfully received until the complete NFCID1 is stored in NFC Forum Device. After the whole NFCID1 reception that NFC Type 2 Tag is put to sleep with *SLEEP_REQ* and NFC Forum Device tries to get NFCID1 from another NFC Type 2 Tag in the field [11].

Chapter 3

Digital design verification and UVM

In general the goal of verification process is to make sure the simulated device completes its task according to the specification. Digital verification has been evolving very quickly together with digital circuits design and because of rapid design complexity, the growth of digital verification has to keep up. Modern hardware verification language needs to model structures also at higher abstraction levels and more similar approach like software programming can be used, because verification modules do not have to be synthesizable [12] [13].

At around 15 years ago, many languages were used for digital circuits verification, like Vera and *e*, but currently the most used language is SystemVerilog. SystemVerilog is based upon Verilog with addition of objected-oriented programming constructs and is IEEE standard since 2005 [12].

First of all, to make the job for verification engineers the most intuitive and their results reusable, each main EDA vendor developed their own methodology for customers using their own tools. The first widely used methodology named Verification Methodology Manual (VMM) had been created by Synopsis, later Mentor Graphics and Cadence introduced Open Verification Methodology (OVM) [13] [14].

These standards along with many others had formed a strong foundation of knowledge and experience, which eventually led to the creation of Universal Verification Methodology (UVM). UVM is derived mostly from OVM, but also uses elements from other standards. Its development is in charge of Accelera Systems - a nonprofit independent organization composed of EDA vendors such as Cadence, Mentor, Synopsis etc [15].

Therefore UVM as an open-source library is widely supported by every major simulation software and is aiming to be the methodology suited for every type of digital design, in regards of size, target technology and design type orientation. However, to reach an efficient level of verifying logic designs, small subset of UVM is really needed. The most important parts are described in the following subsections [16] [13]. At the end of this chapter in Section 3.6 term coverage is defined, so it could be used later in creating complete verification component.

3.1 UVM testbench basics

UVM Testbench is constructed in layered, object-oriented way, which empowers division of work to be done. Modular architecture specifies functions each component is responsible for and that also enables easy reusability [13].

The lowest layer is RTL (or gate-level) description of a Device Under Test (DUT), which communicates with transactor layer through pin-level activity on one or more virtual interfaces. Transactor layer contains monitors and drivers and serves as a conversion layer between the pin-level signals and the transactions. These transactions are then passed on to testbench layer above, which controls the flow of the test and generates input stimulus [13].

All the modules are connected between themselves and encapsulated within one UVM Environment, which is configured and built in UVM Test. UVM Test is therefore the starting point for the build process and apart from the building and configuring environment its job is to specify and apply the constrained-random stimulus [13].

UVM Components in UVM Testbench are instances of UVM Classes, which are derived from *uvm_component class*. These components are part of the testbench hierarchy for the whole duration of the test, while sequences are not (they are extended from *uvm_sequence class*). UVM also implements a lot of other classes and macros, for example useful UVM messaging system, which displays messages in consistent format inside UVM testbench, and much more (see [17]).

3.2 UVM Factory

The UVM Factory creates UVM objects and components and enables to substitute them by derived objects and components without changing the testbench structure. This way the change of sequence behaviour can be changed, or component might be replaced by its newer version [13] [18].

Proper coding must be followed to ensure factory functionality. Firstly, components and object are registrated with corresponding registration macros ‘*uvm_component_utils* and ‘*uvm_object_utils*. The next step is to use a factory constructor with constructor defaults. Subsequently, components are created during the build phase with *create()* method [13] [18].

3.3 UVM Configuration Database

Components and objects may share resources between themselves (by resource is meant any piece of information), and recommended way do to that is *uvm_config_db*. *uvm_config_db* accesses resource database by using two simple methods - *uvm_config_db::set* and *uvm_config_db::get*. There are no limitations on the type of information, and classic examples of using *uvm_config_db* are passing virtual interfaces from DUT domain to the test and passing configuration objects through the testbench hierarchy [17] [13].

3.4 UVM Phases

UVM uses phases to ensure consistent execution flow, and they can be divided into three main groups. UVM phase execution starts by calling *run_test()* method usually from the inside of the top module block. The job of the first group of phases - build phases, is to construct, configure and connect the testbench component hierarchy.

After build phases follow run time phases, in which stimulus is generated and applied to DUT. There are also parallel run-time phases, which are executed together with run phase and allow to specify task execution in time more precisely. In the end information from scoreboards need to be extracted and evaluated, and this task is reserved for clean up phases. During the clean up phases no simulation time need to be used, therefore they are implemented as functions [13].

3.5 UVM Environment

UVM Environment encapsulates all the verification components targeting the DUT, usually they are configurable through data passed from *uvm_config_db* [17].

3.5.1 UVM Sequencer and Sequences

The UVM Sequencer works at testbench layer and controls the flow of UVM Sequence items generated by one or more UVM Sequences to UVM Driver. Sequences are objects that have necessary information for generating stimulus. Unlike the UVM Sequencer, there are not part of the testbench hierarchy, and can be described as a transient object. This means after sequence execution it can be discarded and testbench moves on to next step of UVM Test [13].

3.5.2 UVM Scoreboard

The UVM Scoreboard collects the input and output sequences of DUT through the UVM Agents analysis ports and checks if the DUT behaviour is correct. The check is done by comparison of DUT response and expected output generated by a reference model implemented in the testbench [17].

3.5.3 UVM Driver

Inputs to DUT are on a pin-level basis, that means the transactions passed from the UVM Sequencer must be converted to be able to communicate with DUT via a virtual interface. Drive can also act as a "responder" - it reacts to pin-level activity in the virtual interface and passes the information to the sequence, which then sends a response transaction back to the driver to complete the communication [17] [13].

3.5.4 UVM Monitor

The UVM Monitor has a reverse task as a driver in the testbench - it converts pin-level activity back to transactions. Homewer, the monitor behaviour should be passive only and cannot affect the DUT in any way. A monitor just samples the DUT activity and passes the recognized transactions to the other parts of the testbench. To do so, a monitor must implement protocol rules and look for recognizable patterns in the virtual interface [13].

3.5.5 UVM Agent

All UVM Components dedicated to one logical interface (for example USB) are grouped within one hierarchical component - the UVM Agent. A common UVM Agent consists of an UVM Driver, an UVM Monitor and an UVM Sequencer, but might also contain some other components, for example coverage collector [17].

3.6 SystemVerilog Coverage

Coverage should be the part of designed Universal Verification Component (UVC), therefore it is important to explain its purpose. It can be defined as percentage of verification objectives that have been covered. Basically, the coverage measures tested and untested parts of the design [19].

Two types of coverage may be used. The first one is called code coverage, which measures amount of the code in the design (blocks, lines, state machines) that is tested. Code coverage is measured by simulator and the collection needs to be turned on during the configuration of simulation run [19].

Functional coverage is manually implemented by verification engineer and measures the amount of design specification that is already tested. [20] [19]. Functional coverage uses SystemVerilog *covergroup* construct and can be collected using one common component in the testbench or inside the component where the data are present.

Coverage is used as a central part of CDV - Coverage Driven Methodology. This approach uses test cases only to steer the contrained random sequences toward 100 % coverage. The coverage is therefore the main measure of successful execution of the verification plan, which results in better traceability [21]. CDV approach was later improved by Cadence and labeled by name Metric Driven Verification (MDV). It is based on metrics collections and can be divided into 4 steps executed continuosly until the verification is complete [22].

These 4 steps are:

plan creation of the verification plan, a document specifying the requierements and way of verifying them,

construct implementation of the verification environment,

execute all the test cases are run and results are checked,

measure and analyze coverage data are mapped to the verification plan and results are analyzed. Cadence has its own tool called vManager, which can fire the regression and directly correlate results with the verification plan [22] [23].

Part II

Practical Part

Chapter 4

Verification plan

Verification plan proposal is the first step in creating well structured and reliable verification component. In the following sections of this chapter each aspect of UVC preparation will be described. As it was previously stated, verification component is implemented by using SystemVerilog language and Universal Verification Methodology.

4.1 Requirements

List of requirements has to include all the rules from the design and standards specifications that should be covered by verification. Well written requirements list is therefore an essential part of verification process. Requirements for covering NFC Type 2 Tag platform are based on all NFC specifications [1] [7] [2] [11] together with original ISO 14443 standards, namely [8] and [6].

Requirements are divided into 5 main groups, each requirement has its own description, origin, priority and method, which is used for covering the requirement. This can be done by checker, test case, coverage points or by their combination.

4.1.1 Physical layer requirements

First group of requirements Table 4.1 covers physical aspects of communication, which DUT must follow. This group is solely covered by assertions in *nfc2_t2r_monitor* component, which decodes the response from DUT. Physical layer has the highest priority to be verified.

Requirement	**Specification**	**Priority**	**Checkers/ Coverage/ Scenario (CH/CV/SC)**
HF_PHY_T2R_1	NFC Digital 4.1.2 NFC Digital 4.1.4 NFC Digital 4.2.2 NFC Digital 4.	1	CH
HF_PHY_T2R_2	Digital 4.1.2 NFC Digital 4.1.4 NFC Digital 4.2.2 NFC Digital 4.	1	CH
HF_PHY_T2R_3	NFC Digital 4.1.2 NFC Digital 4.1.4 NFC Digital 4.2.2 NFC Digital 4.	1	CH

Table 4.1: Requirements - physical layer

HF_PHY_T2R_1

Verify that for each response DUT uses OOK f_s subcarrier modulation with Manchester coding, with $f_s = f_c/16 \cong 847.5kHz$ [1].

HF_PHY_T2R_2

Verify that each DUT response frame starts with SoF sequence and ends with EoF sequence [1].

HF_PHY_T2R_3

Verify that standard and bit oriented SDD frames contain correct odd parity and particular responses (SEL_RES, READ_RES) have valid end of data field (EoD - CRC16) [1].

4.1.2 Timing requirements

Another aspect of DUT behaviour is following timing specifications from Table 4.2. These rules also have the highest priority and each is covered by checkers.

Requirement	**Specification**	**Priority**	**Checkers/ Coverage/ Scenario CH/CV/SC**
HF_PHY_T2R_4	NFC Digital 4.10	1	CH
HF_PHY_T2R_5	NFC Digital 4.10	1	CH
HF_PHY_T2R_6	NFC Digital 9.9	1	CH
HF_PHY_T2R_7	NFC Digital 9.9	1	CH
HF_PHY_T2R_8	NFC Digital 9.9	1	CH

Table 4.2: Requirements - Timing specifications

HF_PHY_T2R_4

DUT shall not produce any disturbance during at least $t_n n_m in$ time to $t_n n$ time before the response start. Value of $t_n n_m in$ differs for possible last bits [1]. Detectable disturbance is not defined in this standard, but for digital output signal of DUT is no change on output allowed besides DUT response to command. That is why this requirement is covered by *nfc2_r2t_monitor* structure, which does not allow any unknown patterns and always tries to create a *nfc2_base_res* item, otherwise it reports *'uvm_error*.

HF_PHY_T2R_5

DUT shall send the response at time $FDT_{A,LISTEN,MIN}$ time or after - this depends on the type of command and last V_2 modulation. For group of commands - ALL_REQ, SENS_REQ,, SDD_REQ, SEL_REQ response must be sent at specific point in time equal to $FDT_{A,LISTEN,MIN}$ with tolerance of $-1/f_c to (0.4\mu s + 1/f_c)$, where

$FDT_{A,LISTEN,MIN,log,0} = n * (128/f_c) + 20/f_c = 1172/f_c, where n = 9$, $FDT_{A,LISTEN,MIN,log,1} = n * (128/f_c) + 84/f_c = 1236/f_c, where n = 9$.

Correct $FDT_{A,LISTEN}$ time for each command is verified by checkers in *nfc2_scoreboard*.

HF_PHY_T2R_6

For READ command response must be sent after $FDT_{A,LISTEN,MIN}$ from HF_PHY_T2R_5 and at least 5 ms from the last V_2 mod, while n must be integer value equal or higher than 9. This requirement is verified with checker inside *nfc2_scoreboard*.

HF_PHY_T2R_7

For WRITE command response must be sent after $FDT_{A,LISTEN,MIN}$ from HF_PHY_T2R_5 and at least 10 ms from the last V_2 mod, while n must be integer value equal or higher than 9. This requirement is verified with checker inside *nfc2_scoreboard*.

HF_PHY_T2R_8

SECTOR SELECT response must be sent after $FDT_{A,LISTEN,MIN}$ from HF_PHY_T2R_5 and at least 1 ms from the last V_2 mod, while n must be integer value equal or higher than 9. This requirement is verified with checker inside *nfc2_scoreboard*.

4.1.3 Tag state requirements

Third group defines DUT behaviour in each state from NFC Type 2 Tag Platform State Machine. For verification of this group combination of cross coverage and directed test cases shall be used. These requirements have priority 2, which means they should be verified, but will be implemented after priority 1 requirements. Due to the larger required space, the table for these requirements is omitted, but all requirements are derived from Figure 4.1 and each Tag state has its own requirement [2].

4.1.4 Commands requirements

Response of DUT for each command needs to be verified from logical standpoint. This is implemented by modeling Tag behaviour and predicting the response according to the current state of the Tag derived from Figure 4.1. This

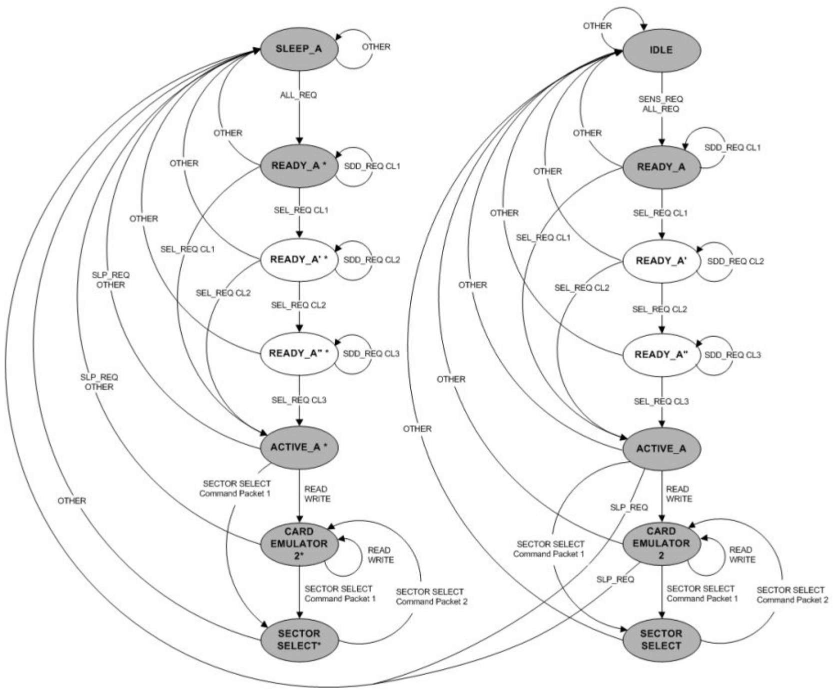


Figure 4.1: Type 2 Tag Platform State Machine [1].

predicted item is later compared to captured item from *nfc2_t2r_monitor* in the *nfc2_scoreboard* by using checkers.

Every command has one requirement, in which correct response has to be verified in the proper state. These requirements are not present in Table 4.3. The requirements in Table 4.3 cover command variable fields and special scenarios.

HF_SDD_REQ_2

Verify DUT response for each valid combination of *SEL_PAR*, *NFCID1* and *CL* fields in *SDD_REQ* command.

HF_SDD_REQ_3

Verify that DUT ignores *SDD_REQ* command with invalid combination of *SEL_PAR* and *NFCID1* fields and transitions to *IDLE/SLEEP_A* state.

Requirement	**Specification**	**Priority**	**Checkers/ Coverage/ Scenario CH/CV/SC**
HF_SDD_REQ_2	NFC Digital 4.7	2	CV/SC
HF_SDD_REQ_3	NFC Digital 4.7	2	SC
HF_SDD_REQ_4	NFC Digital 4.7	2	SC
HF_SEL_REQ_2	NFC Digital 4.8	2	SC
HF_SEL_REQ_3	NFC Digital 4.8	2	SC
HF_SLP_REQ_1	NFC Digital 4.9	2	SC
HF_READ_2	NFC Type 2 Op. Spec. 5.1, 5.4	2	CV/SC
HF_WRITE_2	NFC Type 2 Op. Spec. 5.2, 5.4	2	SC
HF_WRITE_3	NFC Type 2 Op. Spec. 5.2, 5.4	2	SC
HF_WRITE_4	NFC Type 2 Op. Spec. 5.2, 5.4	2	SC
HF_SEC_S_1	NFC Type 2 Op. Spec. 5.3, 5.4	2	CH
HF_SEC_S_2	NFC Type 2 Op. Spec. 5.3, 5.4	2	SC

Table 4.3: Requirements - Commands coverage and scenarios

HF_SDD_REQ_4

Verify that DUT ignores *SDD_REQ* command with higher *CL* field than it actually contains and transitions to *IDLE/SLEEP_A* state.

HF_SEL_REQ_2

Verify that DUT ignores *SEL_REQ* command with higher *CL* field than it actually contains and transitions to *IDLE/SLEEP_A* state.

HF_SEL_REQ_3

Verify that DUT ignores *SEL_REQ* command with incorrect *NFCID1* field for each *CL* field and transitions to *IDLE/SLEEP_A* state.

HF_SLP_REQ_1

Verify successfull *SLP_REQ* command execution by issuing *SENS_REQ* and *ALL_REQ* sequence. DUT shall not respond to *SENS_REQ* but shall respond to *ALL_REQ* and shall transition to *READY_A** state.

HF_READ_2

Verify that DUT sends *NACK* response for *READ* command with block number out of memory space and transitions to *IDLE/SLEEP_A* state.

HF_WRITE_2

Verify that DUT sends *NACK* response for *WRITE* command with block number out of memory space and for blocks 0 and 1 (read-only blocks) and transitions to *IDLE/SLEEP_A* state.

HF_WRITE_3

Verify that DUT locks correct Data blocks from Static Memory Structure after *WRITE* command to block 2.

HF_WRITE_4

Verify that blocks 2 and 3 are irreversible after *WRITE* command - after the write of logic 1 cannot be changed back to logic 0.

HF_SEC_S_1

Verify that DUT issues *NACK* response after *SECTOR SELECT* command packet 1 in *ACTIVE_A/CARD_EMULATOR_A/ACTIVE_A*/ CARD_EMULATOR_A** states and transitions to *IDLE/SLEEP_A* state.

HF_SEC_S_2

Verify that DUT ignores*SECTOR SELECT* command packet 2 and transitions to *IDLE/SLEEP_A* state.

4.1.5 Negative scenarios

Cases labeled as negative scenarios drive the DUT with purposely invalid transactions. Types of invalid injections, which will be verified are sending command before allowed $FDT_{A,POLL}$ time, invalid parity bits, CRC, or completely missing these fields. These requirements should be implemented with their own directed test cases and have priority 3 - could be verified, if the schedule allows it.

4.2 Verification strategy

Verification uses constraint random verification approach and DUT is accessed as a black-box. This way the UVC shall be easily reusable and independent of actual RTL design.

Completeness should be measured by number of requirements implemented and covered. Implemented requirement means that it is linked to all selected items - functional coverage, checker and test cases. Covered requirement has 100 % coverage from all corresponding items.

Test cases should be implemented in two ways - as a constrained random tests with minimum constraints to reach coverage goals faster, or directed approach can be used for specific scenarios. All the test cases are shared for RTL and gate-level simulation.

UVC to be designed should be implemented in the most customizable way. Testbench environment and its components should have their configuration files, where their functions could be turned off.

The goal of verification is to achieve 100 % code coverage and functional coverage with all test cases without any reported error.

4.3 Tools

Main software used for simulations is Cadence Incisive 15.20-s009 For viewing coverage extracted data Cadence Integrated Metrics Center 15.20-s009 is used.

Cadence Incisive is called by user-written Perl script *compile_script*, which has as a parameter *-t* test case which will be run. This script passes all the necessary arguments (e.g. enables coverage collection, Graphics User Interface - GUI, and specifies UVM message level) to common *irun* command that parses and executes them.

4.4 Checkers implementation

All checkers are implemented as immediate assertions located in passive components, i.e. nfc2_t2r_monitor and nfc2_scoreboard. Assertions shall be exclusively labeled and use this template:

```
assert_<unique_name> :assert(<condition>) begin
   //Pass message
   `uvm_info(get_type_name(), $sformatf("%m :
   Information %d", data), <verbosity>)
end
else begin
   `uvm_error("get_type_name()", $sformatf("%m :
   Information : %d", data))
end
```

4.5 Coverage implementation

Functional coverage is implemented inside relevant components with coverage groups containing cover points and cross coverage. For each covergroup this template is used:

```
covergroup cov_grp_<unique_name>;
  cov_<unique_name>          : coverpoint <item_name> {
    //Bins are created automatically or manually
  }
  cross_<unique_items_name>     : cross <item_name_1>,
  <item_name_2> {
    // If needed, some bins may be ignored
    ignore_bins ignore_these_<items> =
    binsof(<item_name_1>) intersect {<val_list1>} ||
    binsof(<item_name_2>) intersect {<val_list2>};
  }
endgroup
```

4.6 Test cases implementation

Test cases shall be derived from *nfc2_test_base* class, which creates the *nfc2_env* verification environment, *nfc2_env_cfg* environment configuration object instance and initializes it. Derived test case modifies the properties in configuration and starts a sequence.

4.7 UVM usage

Designed UVC uses the set of UVM macros and classes described in, namely:

- all UVC components are derived from the most suitable UVM class and use their inherited methods,
- *uvm_config_db* database is used for passing necessary objects between components,
- UVM report macros *'uvm_info*, *'uvm_warning*, *'uvm_error* and *'uvm_fatal* are used most of the time for reporting. Message verbosity is altered by *"+UVM_VERBOSITY=:* option and can be set to UVM_DEBUG, UVM_FULL, UVM_HIGH, UVM_MEDIUM, UVM_LOW or UVM_NONE.

Chapter 5

Implementation of UVC for NFC Type 2 Tag

Designing UVC for NFC Type 2 Tag Platform is the next task after preparation of verification plan, whom content needed to be taken into consideration. Before starting to write the code, it is better to divide the particular tasks and propose a system-level design of solution.

As it was previously mentioned, the main goal of the UVC is to verify DUT's full functionality according to NFC Type 2 Tag Platform. Required UVC should therefore support all protocol commands and corresponding responses and know how to modulate and demodulate communication channels from DUT interface. In the verification plan there are also requirements, which need invalid sequences or valid command in incorrect time. Thus it is essential not only be able to communicate with DUT using the protocol correctly, but also to inject the protocol errors. This is achieved by properly written command classes and compatible driver.

This chapter will describe UVM components and objects that take care of driving the connected DUT, collecting the functional coverage, and predicting and verifying the DUT responses. Base classes of NFC commands and responses will be described before the UVC analysis on a system level, so one can understand the conception of how they are used in the UVC.

5.1 Command and response items

Both NFC command and response classes inherit fields and methods from their respective base classes, and carry all the basic properties. If the command or response is more complex, additional fields or methods are implemented.

5.1.1 *nfc2_base_cmd* class fields

Base item class for commands *nfc2_base_cmd* extends *uvm_sequnce_item* class and contains these fields:

```
class nfc2_base_cmd extends uvm_sequence_item;
  //fields
  rand bit crc_present;
  rand bit cmd_len_valid;
  rand bit crc_valid;
  rand bit parity_ok;
  rand bit sof_present;
  rand bit eof_present;
  rand nfc2_frame_format_e nfc2_frame_format_i;
  rand nfc2_bit_rate_e nfc2_bit_rate_i;
  rand bit[6:0] nfc2_pause_a_len;
  rand bit[7:0] nfcid1_len;
  rand nfc2_wait_after_res_e nfc2_wait_after_res_i;
  nfc2_byte_stream_q tx_stream;
  nfc2_command_e nfc2_command_i;
  int res_max_wait_time;
  bit[7:0]  cmd_code;
  bit[15:0] crc_i;
  time cmd_end_time;
  //...
endclass : nfc2_base_cmd
```

Unusual datatypes from the field declaration are defined in the common SystemVerilog file together with all required constants and datatypes used in the test bench. *nfc2_byte_stream_q* is a queue of bytes and holds the payload that driver processes based on the other fields. *nfc2_command_e* is enumerative type of all supported NFC commands, *nfc2_frame_format_e* similarly holds the possible frame formats, *nfc2_bit_rate_e* bit rates and *nfc2_wait_after_res_e* options for issuing another command before or after allowed $FDT_{POLL,MIN}$ or GT_A time (see Section 2.1.6).

All *rand* fields are properly constrained by using *soft* construct, so in case of negative scenario the constraint might be violated.

nfc2_base_cmd class also implements basic functions for randomization, computing CRC16, converting queue to transaction and vice versa and printing functions for reporting in simulators console. Another characteristics are later implemented in classes for specific commands.

5.1.2 *nfc2_base_res* class fields

Base item class for Tag responses *nfc2_base_res* is also extented from *uvm_sequnce_item* class and contains these common fields:

```
class nfc2_base_res extends uvm_sequence_item;
  //fields
  rand bit crc_valid;
  rand bit crc_present;
  rand bit parity_ok;
  rand bit res_len_valid;
  rand nfc2_frame_format_e nfc2_frame_format_i;
  rand nfc2_bit_rate_e nfc2_bit_rate_i;
  rand bit[7:0] nfcid1_len;
  time res_start_time;
  int unsigned time_res;
  nfc2_command_e nfc2_command_i;
  nfc2_response_e nfc2_response_i;
  nfc2_byte_stream_q rx_stream;
  bit[15:0] crc_i;
  //...
endclass : nfc2_base_res
```

These fields are very similar to *nfc2_base_cmd* fields, but apart from that there is information about the type of response in *nfc2_response_e* enumerative type and response start time for validating timing.

nfc2_base_res class also implements basic functions for randomization, computing CRC16, converting *rx_stream* queue to transaction and vice versa, printing functions for reporting in simulator console and checking correct length of response.

5.2 System-level design of the testbench

Division of the testbench (TB) to modules is displayed on Figure 5.1. Names of the modules all start with shortcut *nfc2_*. In the *nfc2_tb_top* module selected testcase together with DUT module are created, also *virtual interface* connecting DUT to the testbench is created and passed into the *nfc2_env* environment.

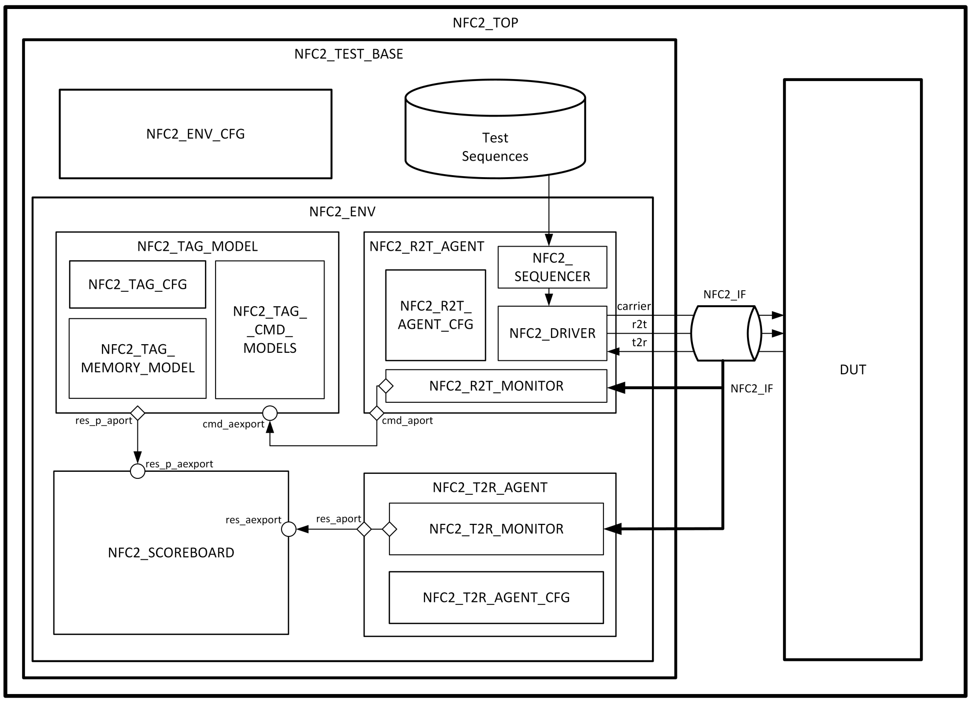


Figure 5.1: System level of implemented testbench.

After that the selected test case is launched and build of the *nfc2_env* and *nfc2_env_cfg* begins. This environment contains 4 components:

- 2 agents designated for transmitting and receiving *logic* signals from the interface,
- *nfc2_tag_model* modeling DUT NFC Type 2 Tag behaviour and predicting the Tag response,
- *nfc2_scoreboard* verifying DUT response by comparing it to *nfc2_tag_model* predicted item.

Environment *nfc2_env* also has to connect these components during *connect_phase* and for that uses *uvm_analysis_port #(<item_name>)*. In this instance three analysis ports are needed. Analysis port *cmd_aport* exports captured command from *nfc2_r2t_monitor* to *nfc2_tag_model*, analysis port *res_aport* passes captured DUT response from *nfc2_t2r_monitor* to *nfc2_scoreboard* and the last analysis port *res_p_aport* transports predicted response from *nfc2_tag_model* to *nfc2_scoreboard*.

5.2.1 Configuration objects

Object *nfc2_env_cfg* holds all configurable settings of the UVC. Inside this object another three configuration files are created - one for each agent and one configuring specific DUT parameters for *nfc2_tag_model*. These objects are passed on to *uvm_config_db* and relevant components can access them by using *get* method.

Configuration objects for the agents specify if the checkers inside its monitors are applied and whether the component is active or passive.

Object *nfc2_tag_cfg* configures all the DUT properties needed for the implemented test bench, which includes:

- number of Data memory blocks,
- response for *SENS_REQ* and *ALL_REQ* commands,
- maximum cascade level,
- mask for *SEL_RES* response,
- information about where the NFCID1 is stored in memory,
- coverage enabling bit.

5.2.2 *nfc2_r2t_agent* and *nfc2_sequencer*

nfc2_r2t_agent encapsulates three components: *nfc2_sequencer*, *nfc2_driver* and *nfc2_r2t_monitor*. Its main task is to instantiate, create and connect them in corresponding phases. Besides that it gets the *virtual interface* from the *uvm_config_db* and assigns it to nfc2_driver.

nfc2_sequencer code is very simple, it just needs to register itself within the UVM factory, all the other methods are inherited from *uvm_sequencer* class. The sequences for driving are given from the test case and are requested by *nfc2_driver* to process them.

5.2.3 *nfc2_driver*

Driver *nfc2_driver* has 2 main roles during the *run_phase*, and that is generating the f_c clock and processing the items given by *nfc2_sequencer* and driving the DUT with that item after the conversion to logical level according to Modified Miller coding from Section 2.1.2. These tasks are done in parallel by using SystemVerilog *fork* construct:

```
fork
  clock_gen(fc_half_per);
  forever begin
    seq_item_port.get_next_item(seq_req_inst);
    process_item(seq_req_inst);
    detect_res_end(seq_req_inst.res_max_wait_time,
    seq_req_inst.nfc2_wait_after_res_i);
    seq_item_port.item_done();
  end
join
```

Implemented *nfc2_driver* uses the relevant item fields (e.g. *parity_ok*, *pause_a_len*) during the whole command processing. It has separate functions for each NFC frame format, and adds parity bits if necessary. *nfc2_driver* component uses set of functions for modulating the DUT input port, which are based on f_c clock cycles.

Method *detect_res_end* after *process_item()* implements basic detection of the DUT response end, and sets time delay according to item field *nfc2_wait_after_res_i*. Delay can be smaller than mandatory time FDT_POLL, MIN, always minimal or randomly generated.

5.2.4 *nfc2_r2t_monitor*

Component monitoring DUT input port was implemented, so each component is independent. That way if now used *nfc2_driver* is disconnected, another module can drive the DUT. Also this *nfc2_r2t_monitor* checks and validates *nfc2_driver* functionality. Main task of *nfc2_r2t_monitor* is running in *forever* loop, where it detects SoF sequence, captures the data bits until the EoF sequence occurs. Command recognition starts after this process, *nfc2_r2t_monitor* first checks amount of bits captured, and looks for the command code match. Based on captured command, CRC16 field might be checked, and correct method to validate command lenght is used. If the frame format is not short, *nfc2_r2t_monitor* also checks parity bits. For timing validation purposes *nfc2_r2t_monitor* stores last modulation time to item. In the end *nfc2_r2t_monitor* writes captured item to analysis port *cmd_aport* and looks for another command SoF.

5.2.5 nfc2_t2r_agent and nfc2_t2r_monitor

Agent *nfc2_t2r_agent* incorporates only the Tag monitor *nfc2_t2r_monitor*, so its duties are to create it, get *virtual interface* from *uvm_config_db* and pass it to the *nfc2_t2r_monitor*.

Tag monitor validates the physical layer of Tag responses alongside the response detection. It works on the same concept as *nfc2_r2t_monitor*. That means detecting the start of frame, catching and decoding the response until the end of response frame sequence appears. *nfc2_t2r_monitor* always tries to recognize the response, but sometimes it is not possible, because information about issued NFC command is needed for definite decision. Nevertheless the parity bits are checked in case of standard or bit oriented SDD frame and data validation is left up to *nfc2_scoreboard*.

5.2.6 *nfc2_tag_model*

Predictor model implements NFC Type 2 Tag Platform behaviour like DUT, but in SystemVerilog language. To do that, *nfc2_tag_model* includes models of behaviour for each supported NFC command and has its own memory model. Memory model is initialized by the same input file as DUT memory, and implements all the features of NFC Type 2 Tag Platform Static Memory Structure. Read and write operations are executed by using *write_block()* and *read_blocks()* functions. At the end of the *write_block()* execution change in static locks settings by *check_static_locks()* function and particular blocks are locked if necessary. The memory model also supports irreversible bits.

Tag predictor reacts to incoming item from *nfc2_r2t_monitor*, and proceeds to response prediction according to *nfc2_command_i* field. At first type of response is predicted by command model. *nfc2_tag_model* stores its own internal state from Tag State Machine and evaluates whether it shall generate the response or ignore the command. Next tag state is also predicted following

Type 2 Tag Platform State Machine [2]. Afterwards the correct response is generated and written to *res_p_aport* analysis port. If the command is ignored, no item is sent through analysis port. *nfc2_tag_model* also in some cases predicts the correct time in which the DUT response has to come. This is done for *SENS_REQ*, *ALL_REQ*, *SDD_REQ* and *SEL_REQ* commands, because they shall be sent at particular time slot. For the rest of the NFC commands computation is carried out in *nfc2_scoreboard*.

5.2.7 *nfc2_scoreboard*

nfc2_scoreboard receives captured and predicted *nfc2_base_res* item and has to compare them. Implemented scoreboard uses simple 1 bit synchronization mechanism, because *nfc2_tag_model* always predicts Tag response before the DUT issues its own response. In case of two consecutive replies from the same source *'uvm_error()* is executed. If the synchronization is intact, *nfc2_scoreboard* compares frame format, payload and validates the timing. If any mismatch occurs, assertion is raised.

Chapter 6

Implemented UVC demonstration

Designed verification component will now be tested with written test case examples. To verify correct functionality of the implemented UVC, already verified and implemented design of HF logic core was provided by ASICentrum company.

6.1 Creating and starting sequence

Sequences are essential part of each test and determine course of the test case. Here is *ALL_REQ* command example of how sequence in test case should be created, built, randomized and run:

```
class test_example extends nfc2_test_base;

  `uvm_component_utils(test_example)

  nfc2_cmd_all_req_seq example_seq;

  function new(string name = "test_example",
  uvm_component parent = null);
    super.new(name, parent);
    example_seq = new;
  endfunction : new

  virtual function void build_phase(uvm_phase
  phase);
```

```
    super.build_phase(phase);
    example_seq = nfc2_cmd_all_req_seq::
    type_id::create("example_seq", this);

    assert(example_seq.randomize() with{
      nfc2_wait_after_res_i == RANDOM_E;
    });
  endfunction : build_phase

  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    example_seq.start(nfc2_env_inst.
    nfc2_r2t_agent_inst.nfc2_sequencer_inst);
    phase.drop_objection(this);
  endtask
endclass : test_example
```

6.2 Example testcases

Multiple constrained-random test cases with original scenarios were written and executed to prove UVC correct functionality. Each of these test cases finished with 0 raised assertions and zero *'uvm_warning*, *'uvm_error* and *'uvm_fatal* messages.

In the following figures from the simulator it can be seen the UVC detection of commands in internal signals *cmd_<name>_detected* from *nfc2_t2r_monitor* and the current state of *nfc2_tag_model* with name *m_state*, while *curr_state_main* is internal signal from the DUT. The first three waveforms are DUT inputs and outputs - *carrier* being the f_c clock, *r2t* being input commands from the UVC and *t2r* DUT responses.

6.2.1 Transition to *ACTIVE_A* state and reading the whole memory

First test case and Figure 6.1 illustrates transition to *ACTIVE_A* state by using *SENS_REQ* command and after that *SEL_REQ* command twice, because DUT is configured for 2 cascade levels. Correct *NFCID1* is read from the UVC memory model, which is created inside *nfc2_test_base* class.

Figure 6.1 together with the rest of the simulation waveform also displays the Tag model functionality. After each issued command in *r2t* signal *m_state* changes exactly in the moment, while the DUT makes the change afterwards.

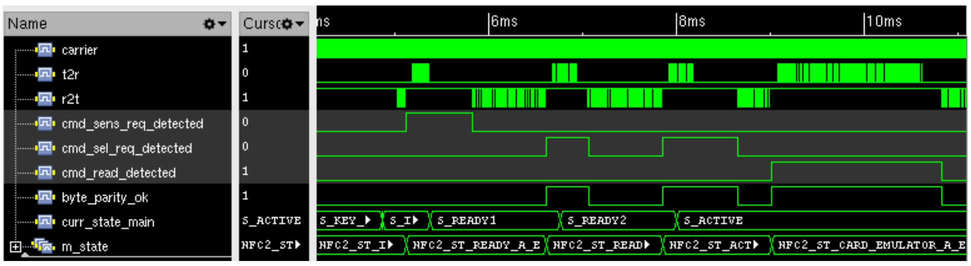


Figure 6.1: Transition to *ACTIVE_A* state and first *READ* command.

In the end of the test case *READ* command with block number out of memory space is issued (see Figure 6.2). DUT shall respond with *NACK* response and test case is complete.

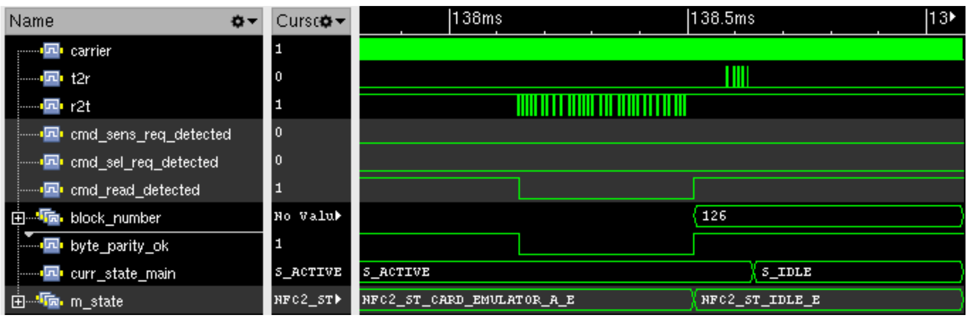
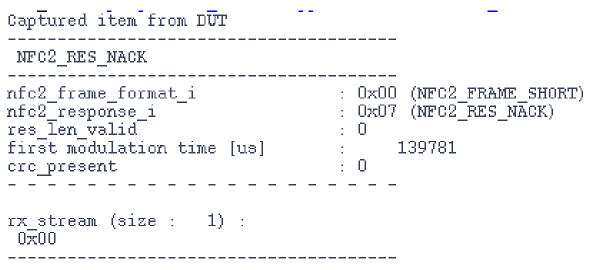


Figure 6.2: Issuing *READ* with higher block number

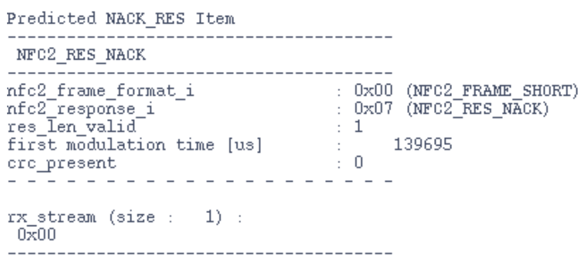
Example of the validation reporting shows item from *nfc2_t2r_monitor* in Figure 6.3 and item from *nfc2_tag_model* in Figure 6.4.



```
Captured item from DUT
---------------------------------------
 NFC2_RES_NACK
---------------------------------------
nfc2_frame_format_i                : 0x00 (NFC2_FRAME_SHORT)
nfc2_response_i                    : 0x07 (NFC2_RES_NACK)
res_len_valid                      : 0
first modulation time [us]         :     139781
crc_present                        : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :   1) :
 0x00
---------------------------------------
```

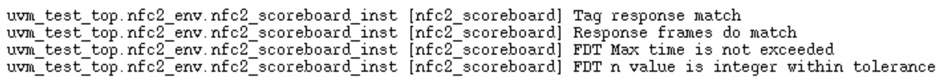
Figure 6.3: Captured *NACK* response

Scoreboard compares the items and reports its results like in Figure 6.5.



```
Predicted NACK_RES Item
---------------------------------------
 NFC2_RES_NACK
---------------------------------------
nfc2_frame_format_i              : 0x00 (NFC2_FRAME_SHORT)
nfc2_response_i                  : 0x07 (NFC2_RES_NACK)
res_len_valid                    : 1
first modulation time [us]       :      139695
crc_present                      : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :    1) :
 0x00
---------------------------------------
```

Figure 6.4: Predicted *NACK* response

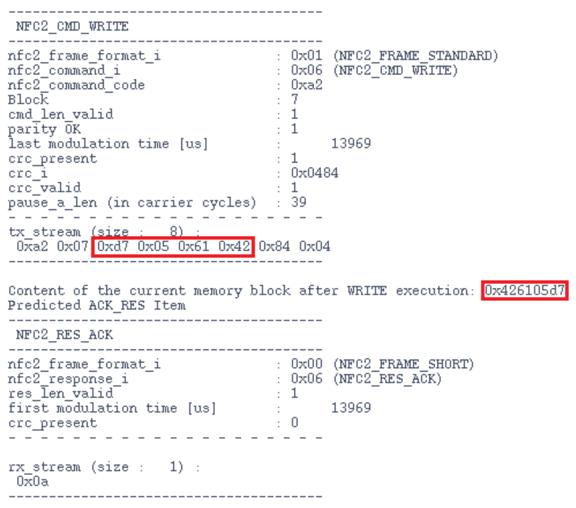


```
uvm_test_top.nfc2_env.nfc2_scoreboard_inst [nfc2_scoreboard] Tag response match
uvm_test_top.nfc2_env.nfc2_scoreboard_inst [nfc2_scoreboard] Response frames do match
uvm_test_top.nfc2_env.nfc2_scoreboard_inst [nfc2_scoreboard] FDT Max time is not exceeded
uvm_test_top.nfc2_env.nfc2_scoreboard_inst [nfc2_scoreboard] FDT n value is integer within tolerance
```

Figure 6.5: Scorebord validation UVM info.

6.2.2 Transition to *ACTIVE_A* state and writing data to Data blocks

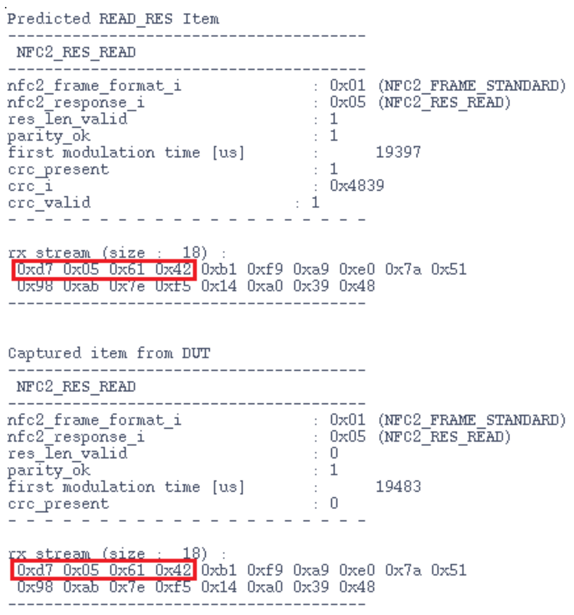
In the second test case *WRITE* command are issued with random data for each Data block. Wheter the data are written correctly is checked by the following *READ* commands. That way the same run of the memory model and DUT memory is proven. The correct write procedure to the memory model is displayed for one block in Figure 6.6 and match with the DUT memory is shown in responses to *READ* command in Figure 6.7. It is important to notice that byte order in the command and the output of the memory model is reversed.



```
---------------------------------------
 NFC2_CMD_WRITE
---------------------------------------
nfc2_frame_format_i              : 0x01 (NFC2_FRAME_STANDARD)
nfc2_command_i                   : 0x06 (NFC2_CMD_WRITE)
nfc2_command_code                : 0xa2
Block                            : 7
cmd_len_valid                    : 1
parity OK                        : 1
last modulation time [us]        :      13969
crc_present                      : 1
crc_i                            : 0x0484
crc_valid                        : 1
pause_a_len (in carrier cycles)  : 39
- - - - - - - - - - - - - - - - - - - -
tx_stream (size :    8) :
 0xa2 0x07 0xd7 0x05 0x61 0x42 0x84 0x04
---------------------------------------

Content of the current memory block after WRITE execution: 0x426105d7
Predicted ACK_RES Item
---------------------------------------
 NFC2_RES_ACK
---------------------------------------
nfc2_frame_format_i              : 0x00 (NFC2_FRAME_SHORT)
nfc2_response_i                  : 0x06 (NFC2_RES_ACK)
res_len_valid                    : 1
first modulation time [us]       :      13969
crc_present                      : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :    1) :
 0x0a
---------------------------------------
```

Figure 6.6: Successful *WRITE* command to block 7.

In the end of the test case *WRITE* command with random block number



```
Predicted READ_RES Item
----------------------------------------
 NFC2_RES_READ
----------------------------------------
nfc2_frame_format_i                  : 0x01 (NFC2_FRAME_STANDARD)
nfc2_response_i                      : 0x05 (NFC2_RES_READ)
res_len_valid                        : 1
parity_ok                            : 1
first modulation time [us]           :      19397
crc_present                          : 1
crc_i                                : 0x4839
crc_valid                          : 1
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :  18) :
 0xd7 0x05 0x61 0x42 0xb1 0xf9 0xa9 0xe0 0x7a 0x51
 0x98 0xab 0x7e 0xf5 0x14 0xa0 0x39 0x48
----------------------------------------


Captured item from DUT
----------------------------------------
 NFC2_RES_READ
----------------------------------------
nfc2_frame_format_i                  : 0x01 (NFC2_FRAME_STANDARD)
nfc2_response_i                      : 0x05 (NFC2_RES_READ)
res_len_valid                        : 0
parity_ok                            : 1
first modulation time [us]           :      19483
crc_present                          : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :  18) :
 0xd7 0x05 0x61 0x42 0xb1 0xf9 0xa9 0xe0 0x7a 0x51
 0x98 0xab 0x7e 0xf5 0x14 0xa0 0x39 0x48
----------------------------------------
```

Figure 6.7: Validation of the previous *WRITE* command with the *READ* command.

out of memory space is issued (see Figure 6.8). DUT shall respond with *NACK* response and transition to *IDLE* state.

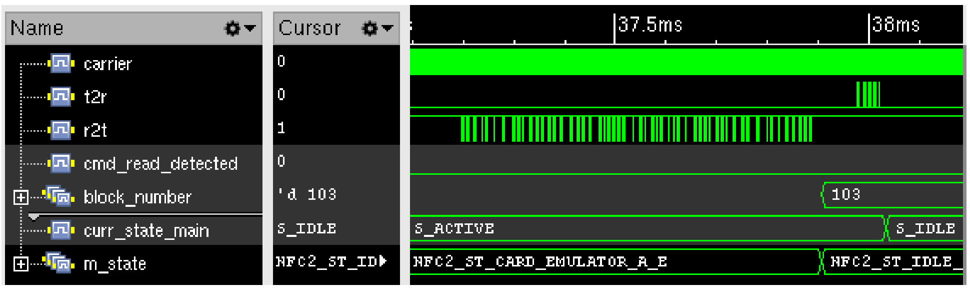
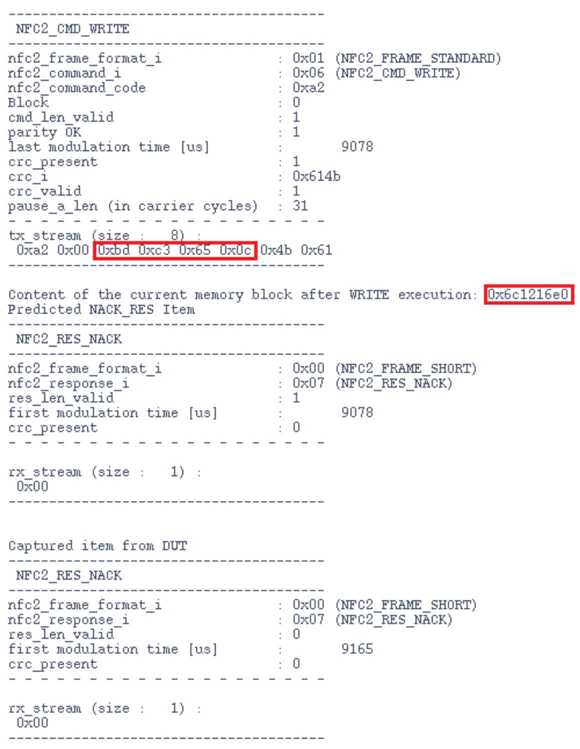


Figure 6.8: Issuing *WRITE* with higher block number

6.2.3 Transition to *ACTIVE_A* state and testing lock bits

This testcase proves that memory model is compliant with Static Memory Data Structure. First sequence is issuing *WRITE* command to block 0 with random data in *ACTIVE_A* state. Block 0 is reserved for manufacturer use and shall be read-only, therefore Tag shall issue *NACK* response. The DUT follows this rule and Figure 6.9 shows that the data in memory model are not changed and *NACK* response is predicted correctly.

The test case continues with locking one block of data by changing one



```
---------------------------------------
 NFC2_CMD_WRITE
---------------------------------------
nfc2_frame_format_i               : 0x01 (NFC2_FRAME_STANDARD)
nfc2_command_i                    : 0x06 (NFC2_CMD_WRITE)
nfc2_command_code                 : 0xa2
Block                             : 0
cmd_len_valid                     : 1
parity OK                         : 1
last modulation time [us]         :       9078
crc_present                       : 1
crc_i                             : 0x614b
crc_valid                         : 1
pause_a_len (in carrier cycles)   : 31
- - - - - - - - - - - - - - - - - - - -
tx_stream (size :   8) :
 0xa2 0x00 0xbd 0xc3 0x65 0x0c 0x4b 0x61
---------------------------------------

Content of the current memory block after WRITE execution: 0x6c1216e0
Predicted NACK_RES Item
---------------------------------------
 NFC2_RES_NACK
---------------------------------------
nfc2_frame_format_i               : 0x00 (NFC2_FRAME_SHORT)
nfc2_response_i                   : 0x07 (NFC2_RES_NACK)
res_len_valid                     : 1
first modulation time [us]        :       9078
crc_present                       : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :   1) :
 0x00
---------------------------------------


Captured item from DUT
---------------------------------------
 NFC2_RES_NACK
---------------------------------------
nfc2_frame_format_i               : 0x00 (NFC2_FRAME_SHORT)
nfc2_response_i                   : 0x07 (NFC2_RES_NACK)
res_len_valid                     : 0
first modulation time [us]        :       9165
crc_present                       : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :   1) :
 0x00
---------------------------------------
```

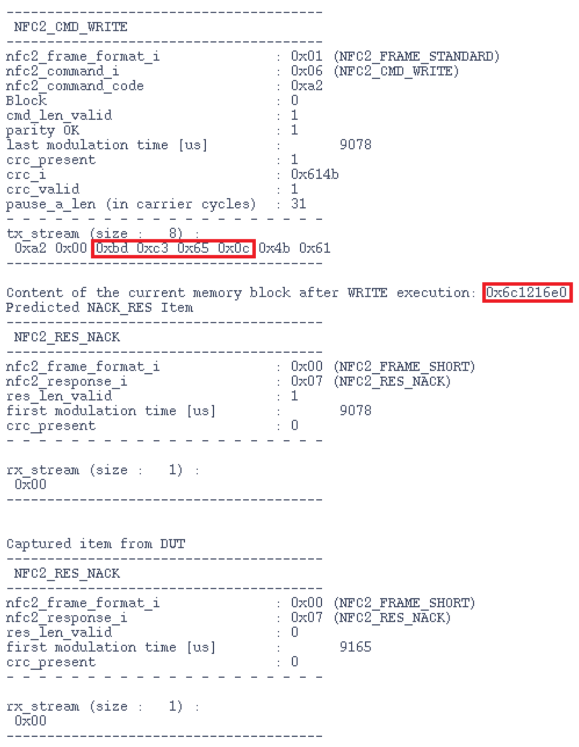
Figure 6.9: Issuing *WRITE* with block number 0.

static lock bit to logic 1. After successful lock the *WRITE* command to that block is performed. Figure 6.10 demonstrates no change of data in the memory model and correctly predicted *NACK* response.

Lock bit already set to logic 1 cannot be changed back to logic 0. This property is tested by trying to write logic 0 back to that bit. Output from the console in Figure 6.11 informs that there was an attempt to change irreversible bit and the value of the lock bit stays at logic 1. Again, the byte order for the labeled data is reversed.

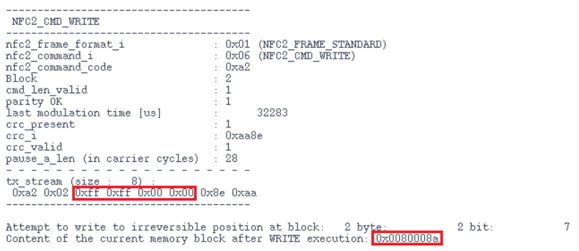
6.2.4 *SLEEP_REQ* command execution

At first transition to *ACTIVE_A* has to be made, then *SLEEP_REQ* has to be executed. After that follows *SLEEP_REQ*, *SENS_REQ* and *ALL_REQ*, and as it is in Figure 6.12, DUT behaves correctly and after *SLEEP_REQ* responds only to *ALL_REQ*.



```
----------------------------------------
 NFC2_CMD_WRITE
----------------------------------------
nfc2_frame_format_i                : 0x01 (NFC2_FRAME_STANDARD)
nfc2_command_i                     : 0x06 (NFC2_CMD_WRITE)
nfc2_command_code                  : 0xa2
Block                              : 0
cmd_len_valid                      : 1
parity OK                          : 1
last modulation time [us]          :       9078
crc_present                        : 1
crc_i                              : 0x614b
crc_valid                          : 1
pause_a_len (in carrier cycles)    : 31
- - - - - - - - - - - - - - - - - - - -
tx_stream (size :    8) :
 0xa2 0x00 0xbd 0xc3 0x65 0x0c 0x4b 0x61
----------------------------------------

Content of the current memory block after WRITE execution: 0x6c1216e0
Predicted NACK_RES Item
----------------------------------------
 NFC2_RES_NACK
----------------------------------------
nfc2_frame_format_i                : 0x00 (NFC2_FRAME_SHORT)
nfc2_response_i                    : 0x07 (NFC2_RES_NACK)
res_len_valid                      : 1
first modulation time [us]         :       9078
crc_present                        : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :    1) :
 0x00
----------------------------------------


Captured item from DUT
----------------------------------------
 NFC2_RES_NACK
----------------------------------------
nfc2_frame_format_i                : 0x00 (NFC2_FRAME_SHORT)
nfc2_response_i                    : 0x07 (NFC2_RES_NACK)
res_len_valid                      : 0
first modulation time [us]         :       9165
crc_present                        : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :    1) :
 0x00
----------------------------------------
```

Figure 6.10: Issuing *WRITE* for previously locked block.



```
----------------------------------------
 NFC2_CMD_WRITE
----------------------------------------
nfc2_frame_format_i                : 0x01 (NFC2_FRAME_STANDARD)
nfc2_command_i                     : 0x06 (NFC2_CMD_WRITE)
nfc2_command_code                  : 0xa2
Block                              : 2
cmd_len_valid                      : 1
parity OK                          : 1
last modulation time [us]          :       32283
crc_present                        : 1
crc_i                              : 0xaa8e
crc_valid                          : 1
pause_a_len (in carrier cycles)    : 28
- - - - - - - - - - - - - - - - - - - -
tx_stream (size :    8) :
 0xa2 0x02 0xff 0xff 0x00 0x00 0x8e 0xaa
----------------------------------------

Attempt to write to irreversible position at block:   2 byte:          2 bit:          7
Content of the current memory block after WRITE execution: 0x0080008a
```

Figure 6.11: Issuing logic 0 for irreversible static lock bit.

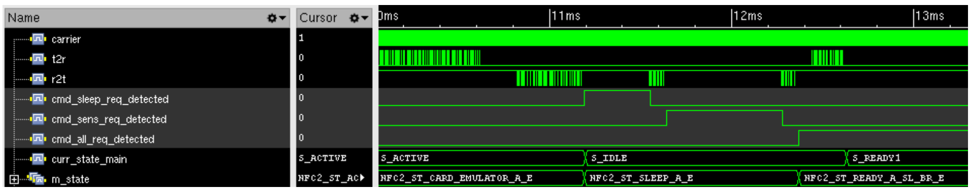


Figure 6.12: *SLEEP_REQ* command verification.

6.2.5 *SECTOR SELECT* command packets 1 and 2

SECTOR SELECT switches to another memory sectors, however DUT contains only 1 memory sector. It follows that to *SECTOR SELECT* command packet 1 DUT shall respond with NACK response and DUT shall ignore the

SECTOR SELECT command packet 2 and transition to *IDLE* state, which Figure 6.13 confirms.

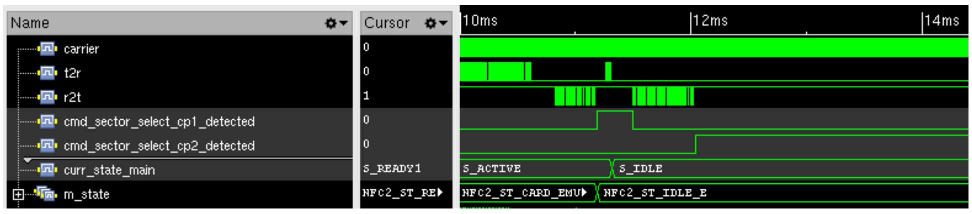


Figure 6.13: *SECTOR SELECT* command packets verification.

6.2.6 *SDD_REQ* for all valid *NFCID1* and *SEL_PAR* combinations

The test case shows the usage of *SDD_REQ* command with all the valid combinations of *SEL_PAR* and *NFCID1* fields. Bit oriented SDD frame recognition and validation was tested this way and UVC finished the DUT testing with 0 errors.

6.2.7 Negative scenarios

In all of these scenarios Tag shall ignore the invalid command and transition to *IDLE* state. First case is Figure 6.14 where *SEL_REQ* is issued in *READY_A* state with all valid fields except CRC16. CRC16 is mandatory, but in this sequence it is omitted and Tag together with the UVC ignores this command.

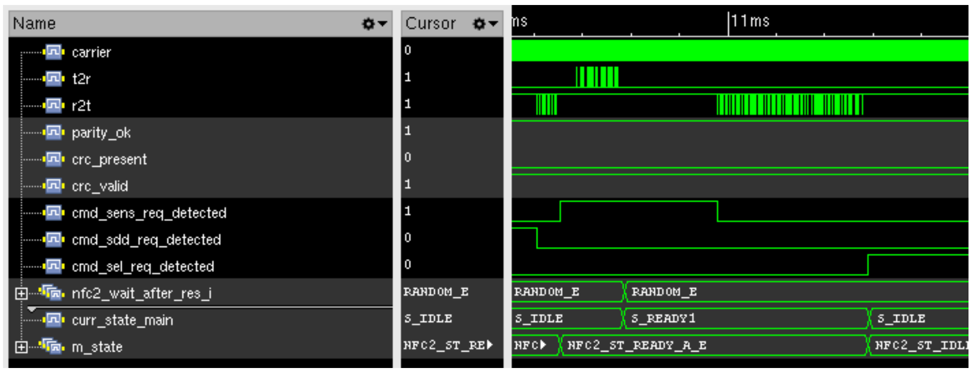


Figure 6.14: *SEL_REQ* without CRC16.

The second example in Figure 6.15 is very similar, but this time the *SEL_REQ* has CRC16 with invalid value. Simulation result is the same.

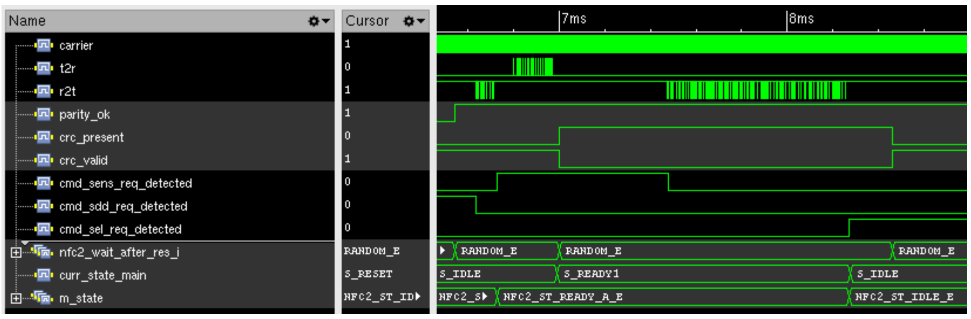


Figure 6.15: *SEL_REQ* with invalid CRC16.

Another sequence in Figure 6.16 consists of transition to *READY_A* state and *SDD_REQ* command with invalid parity bits. Tag and UVC again ignore the command and go to *IDLE* state.

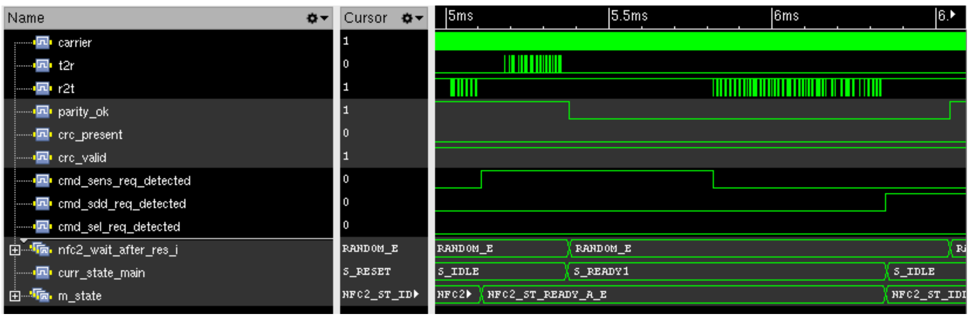


Figure 6.16: *SDD_REQ* with invalid parity bits.

Last example displays option of issuing command before $FDT_{POLL,A}$ time. Simulation waveforms show correct reaction of the DUT and UVC, which both ignore this command.

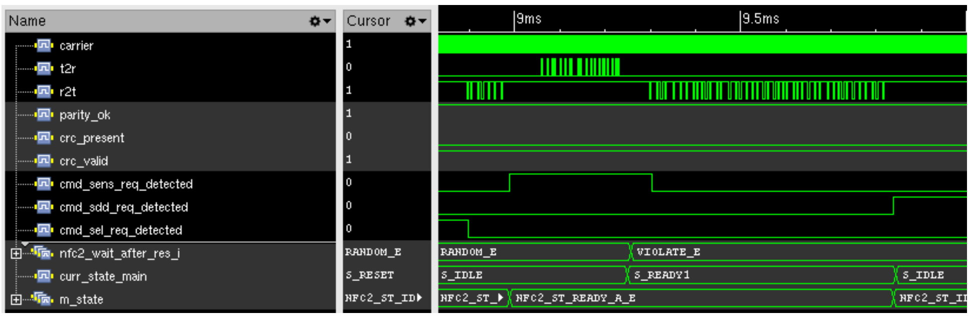
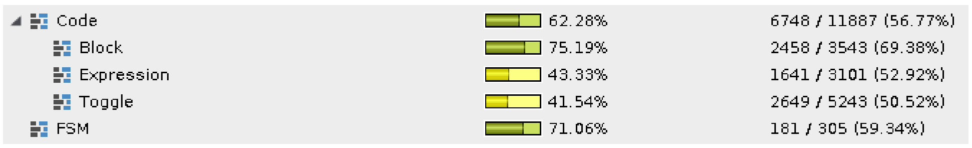


Figure 6.17: *SDD_REQ* issued before allowed $FDT_{POLL,A}$ time.

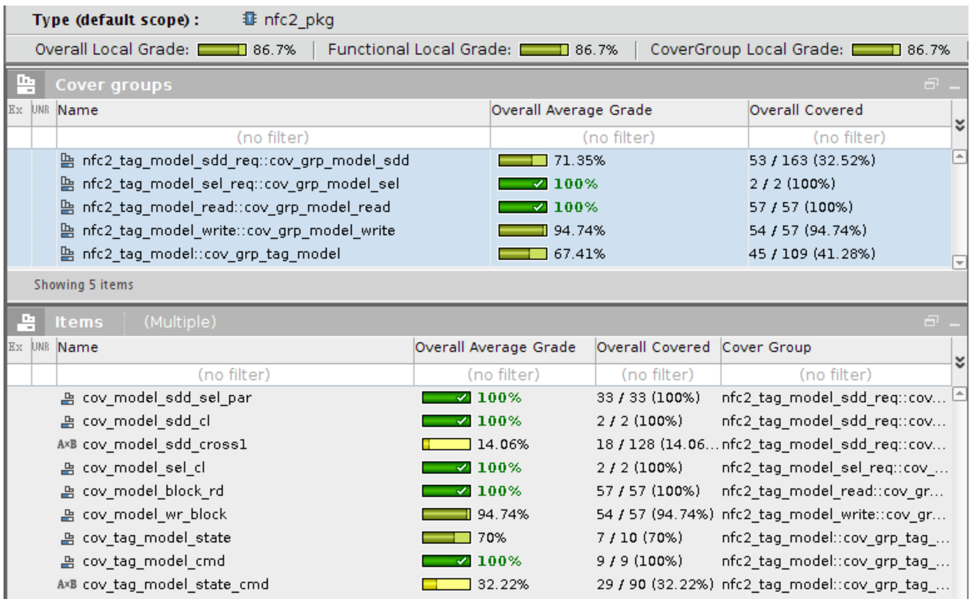
6.3 Coverage sampling

The described test cases from Section 6.2 were also used as an example for collecting code and functional coverage. Integrated Metrics Center report of code coverage in Figure 6.18 shows 62.28 % overall coverage of the DUT source code. The functional coverage inside *nfc2_tag_model* is according to Figure 6.19 covered from 86.7 %. Figure 6.19 also displays percentage of *coverpoints*, that are covered. To declare the DUT as successfully verified, both percentage for code and functional coverage has to be at 100 %. That can be achieved by completing the list of the test cases.



Code	62.28%	6748 / 11887 (56.77%)
Block	75.19%	2458 / 3543 (69.38%)
Expression	43.33%	1641 / 3101 (52.92%)
Toggle	41.54%	2649 / 5243 (50.52%)
FSM	71.06%	181 / 305 (59.34%)

Figure 6.18: Code coverage report after the current set of the test cases.



Type (default scope) : nfc2_pkg

Overall Local Grade: 86.7% | Functional Local Grade: 86.7% | CoverGroup Local Grade: 86.7%

Cover groups

Name	Overall Average Grade	Overall Covered
(no filter)	(no filter)	(no filter)
nfc2_tag_model_sdd_req::cov_grp_model_sdd	71.35%	53 / 163 (32.52%)
nfc2_tag_model_sel_req::cov_grp_model_sel	✓ 100%	2 / 2 (100%)
nfc2_tag_model_read::cov_grp_model_read	✓ 100%	57 / 57 (100%)
nfc2_tag_model_write::cov_grp_model_write	94.74%	54 / 57 (94.74%)
nfc2_tag_model::cov_grp_tag_model	67.41%	45 / 109 (41.28%)

Showing 5 items

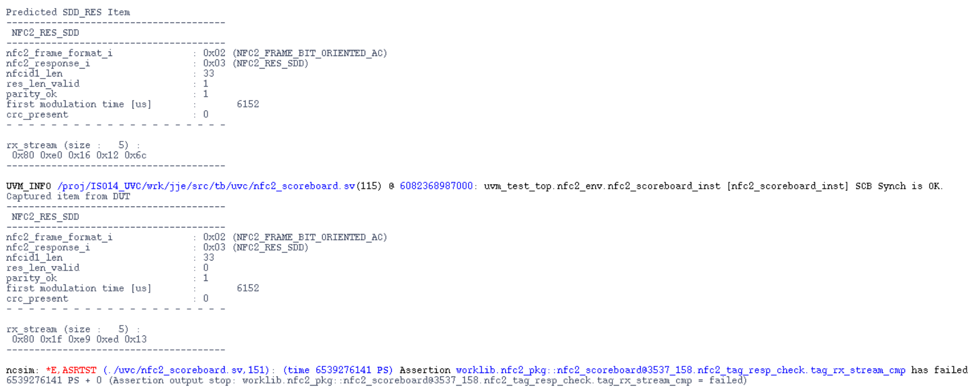
Items (Multiple)

Name	Overall Average Grade	Overall Covered	Cover Group
(no filter)	(no filter)	(no filter)	(no filter)
cov_model_sdd_sel_par	✓ 100%	33 / 33 (100%)	nfc2_tag_model_sdd_req::cov...
cov_model_sdd_cl	✓ 100%	2 / 2 (100%)	nfc2_tag_model_sdd_req::cov...
cov_model_sdd_cross1	14.06%	18 / 128 (14.06...	nfc2_tag_model_sdd_req::cov...
cov_model_sel_cl	✓ 100%	2 / 2 (100%)	nfc2_tag_model_sel_req::cov_...
cov_model_block_rd	✓ 100%	57 / 57 (100%)	nfc2_tag_model_read::cov_gr...
cov_model_wr_block	94.74%	54 / 57 (94.74%)	nfc2_tag_model_write::cov_gr...
cov_tag_model_state	70%	7 / 10 (70%)	nfc2_tag_model::cov_grp_tag_...
cov_tag_model_cmd	✓ 100%	9 / 9 (100%)	nfc2_tag_model::cov_grp_tag_...
cov_tag_model_state_cmd	32.22%	29 / 90 (32.22%)	nfc2_tag_model::cov_grp_tag_...

Figure 6.19: Functional coverage report after the current set of the test cases.

6.4 DUT error detection

Main reason for developing verification environment in general is to detect hidden errors inside the DUT. To show the capability of implemented UVC of detecting bugs, known error was inserted into DUT.



```
Predicted SDD_RES Item
---------------------------------------
 NFC2_RES_SDD
---------------------------------------
nfc2_frame_format_i                : 0x02 (NFC2_FRAME_BIT_ORIENTED_AC)
nfc2_response_i                    : 0x03 (NFC2_RES_SDD)
nfcid1_len                         : 33
res_len_valid                      : 1
parity_ok                          : 1
first modulation time [us]         :       6152
crc_present                        : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :    5) :
 0x80 0xe0 0x16 0x12 0x6c
---------------------------------------

UVM_INFO /proj/IS014_UVC/wrk/jje/src/tb/uvc/nfc2_scoreboard.sv(115) @ 6082368987000: uvm_test_top.nfc2_env.nfc2_scoreboard_inst [nfc2_scoreboard_inst] SCB Synch is OK.
Captured item from DUT
---------------------------------------
 NFC2_RES_SDD
---------------------------------------
nfc2_frame_format_i                : 0x02 (NFC2_FRAME_BIT_ORIENTED_AC)
nfc2_response_i                    : 0x03 (NFC2_RES_SDD)
nfcid1_len                         : 33
res_len_valid                      : 0
parity_ok                          : 1
first modulation time [us]         :       6152
crc_present                        : 0
- - - - - - - - - - - - - - - - - - - -

rx_stream (size :    5) :
 0x80 0x1f 0xe9 0xed 0x13
---------------------------------------

ncsim: *E,ASRTST (./uvc/nfc2_scoreboard.sv,151): (time 6539276141 PS) Assertion worklib.nfc2_pkg::nfc2_scoreboard@3537_158.nfc2_tag_resp_check.tag_rx_stream_cmp has failed
6539276141 PS + 0 (Assertion output stop: worklib.nfc2_pkg::nfc2_scoreboard@3537_158.nfc2_tag_resp_check.tag_rx_stream_cmp = failed)
```

Figure 6.20: DUT error detection in *SDD_RES*

Simulation output in Figure 6.20 shows immediate assertion in the *nfc2_scoreboard* raised, when unexpected response from DUT occured. Buffer for response creation for *SDD_REQ* command was inverted and read from the DUT memory was therefore incorrect, while the *nfc2_tag_model* predicted valid response with the same input memory file.

Chapter 7

Conclusions

This master's thesis's aim was to implement a solution for NFC Type 2 Tag Platform DUT digital verification. At first groundwork for the implementation is set by fundamentals of used standards and software tools. Implemented NFC Type 2 Tag Platform is described, and the next chapter focuses on methods of digital verification, especially UVM library.

The second part moves to more practical part of the job. The verification plan in Chapter 4 outlines the whole process and sums up all the requirements that UVC has to cover. UVC was later designed according to the verification plan and its functionality is demonstrated in Chapter 6 with already functional DUT from ASICentrum company.

All commands from NFC Type 2 Tag Operation Specification are supported, having their own classes and may be easily randomized. UVC implements its own behaviour of Type 2 Tag together with memory model of Static Memory Structure, which may be extended to Dynamic Memory Structure by adding dynamic lock bits in the future. DUT properties may be changed through access to configuration files directly in the test case, and the memory model is initialized from the file. Besides these properties UVC is able to collect functional coverage of all variable commands fields and combination of commands and DUT internal states.

This verification component also collects simulation time data and verifies timing aspects of DUT responses according to issued NFC command. Any mismatch with expected response triggers UVM error message and verification engineer knows immediately about DUT bug, which was displayed in Section 6.4. Apart from this example all the written test cases passed with zero raised assertions and zero UVM error messages, which validates UVC functionality.

UVC was designed with emphasis on automatic response validation, possible configuration changes, easy future extensions and insertion into larger verification environments. It is compliant with Universal Verification Methodolody and fully uses its benefits, thus is compatible with latest digital verification trends and might be used for validating new digital designs. With the complete set of testcases this UVC should be able to cover 100 % of the requirements for NFC Type 2 Operation Specification. The implemented UVC should also be made reset-aware in the future. In conclusion this master's thesis was a very valuable lesson and a great preparation for the future digital verification challenges.

Appendices

Appendix A

Bibliography

[1] NFC Forum. Digital protocol, May 2017. Available at `https://members.nfc-forum.org//specs/`.

[2] NFC Forum. Type 2 tag operation specification, May 2011. Available at `https://members.nfc-forum.org//specs/`.

[3] NFC Forum. What is nfc?, 2019. Available at `https://nfc-forum.org/what-is-nfc/`.

[4] Vedat Coskun, Busra Ozdenizci, and Kerem Ok. The survey on near field communication. *Sensors*, 15:13348–13405, June 2015. Available at `https://www.researchgate.net/publication/278030190`.

[5] Archit Dua. Nfc standards and nfc forum, July 2017. Available at `https://rfid4u.com/nfc-standards-nfc-forum/`.

[6] Iso/iec 14443-3, April 2011. Available at `https://www.iso.org/standard/50942.html`.

[7] NFC Forum. Analog, May 2011. Available at `https://members.nfc-forum.org//specs/`.

[8] Iso/iec 14443-2, September 2010. Available at `https://www.iso.org/standard/50941.html`.

[9] Limor Fried. About the ndef format, May 2015. Available at `https://learn.adafruit.com/adafruit-pn532-rfid-nfc/ndef`.

[10] NFC Forum. Ndef, May 2011. Available at `https://members.nfc-forum.org//specs/`.

[11] NFC Forum. Activity, May 2011. Available at https://members.nfc-forum.org//specs/.

[12] Chris Spear. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2006.

[13] Mentor. *UVM Cookbook*. Mentor, 2018. Available at https://verificationacademy.com/cookbook/uvm.

[14] Stuart Sutherland and Tom Fitzpatrick. Uvm rapid adoption: A practical subset of uvm. *DVCon*, March 2015. Available at https://s3.amazonaws.com/verificationacademy-news/DVCon2015/Papers/dvcon-2015_UVM-Rapid-Adoption-A-Practical-Subset-of-UVM-Paper.pdf.

[15] Uvm (universal verification methodology), 2019. Available at https://www.accellera.org/downloads/standards/uvm.

[16] Doulos. The universal verification methodology, 2019. Available at https://www.doulos.com/knowhow/sysverilog/uvm/.

[17] Universal verification methodology (uvm) 1.2 user's guide, October 2015. Available at https://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[18] Uvm factory, 2019. Available at https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1a/html/files/base/uvm_factory-svh.html.

[19] About systemverilog coverage, 2016-2019. Available at https://www.verificationguide.com/p/systemverilog-coverage.html.

[20] Systemverilog functional coverage, 2016-2019. Available at https://www.chipverify.com/systemverilog/systemverilog-functional-coverage.

[21] Coverage-driven verification methodology, 2005-2019. Available at https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/coverage-driven/.

[22] Sini Balakrishnan. A glimpse on metric driven verification methodology, 2016. Available at http://vlsi.pro/a-glimpse-on-metric-driven-verification-methodology/.

[23] Metric driven verification, 2016-2019. Available at https://www.aldec.com/en/solutions/functional_verification/metric_driven_verification.